Appunti di Informatica 1

Gianluca Rossi

Versione maggio 2011

Indice

Capitolo 1

Algoritmi, macchine e linguaggi di programmazione

Alcuni concetti matematici vengono definiti in maniera assiomatica come ad esempio la radice quadrata del numero x: $\sqrt{x} = y$ se e solo se $y^2 = x$. Questa definizione non ci dice nulla su come calcolare $\sqrt{x}$: dal punto di vista di un matematico essa è ineccepibile ma dal punto di vista di un informatico che vuole progettare un algoritmo per il calcolo di $\sqrt{x}$ è poco utilizzabile. Per fortuna la matematica ci dice anche che se z_0 è un qualunque numero positivo allora

$$\sqrt{x} = \lim_{n \to +\infty} z_n \quad \text{dove } z_n = \frac{1}{2}\left(z_{n-1} + \frac{x}{z_{n-1}}\right).$$

Questo modo alternativo di indicare il numero $\sqrt{x}$ è sicuramente meno intuitivo ma suggerisce un metodo per il calcolo effettivo della radice cercata:

1. Fissa un margine di errore $\epsilon > 0$;
2. Fissa $z = x$;
3. Se $|z^2 - x| \leq \epsilon$ termina e restituisci z

   altrimenti fissa $z = \frac{1}{2}\left(z + \frac{x}{z}\right)$;
4. Torna in 3.

Questa descrizione risulta essere abbastanza formale e priva di ambiguità tanto da poter essere utilizzata da una persona che non conosce cosa sia la radice quadrata di un numero ma sia in grado di eseguire operazioni elementari.

Ricapitolando, abbiamo espresso la stessa conoscenza matematica in due modi diversi: in modo assiomatico ed in modo metodologico. Il secondo ci ha permesso di ricavare un *algoritmo* per il calcolo della radice quadrata di qualsiasi numero positivo x con precisione arbitraria. Il modo metodologico di esprimere la conoscenza è quello tipico dell'informatico che si occupa di progettare algoritmi.

Ora si consideri il problema di determinare se un numero intero $n > 2$ sia primo o meno. La definizione di primalità è metodologica in quanto ci dice che il numero n è primo se e solo se è divisibile solo per 1 e per se stesso. Questo induce un algoritmo che consiste nel verificare, per ogni d compreso tra 2 e $n - 1$, se d divide n. Non appena troviamo tale d possiamo concludere che n non è primo. Viceversa, se tale intero non viene

trovato concludiamo che n è primo. Questo algoritmo nel peggiore dei casi esegue $n-2$ divisioni. Si osservi che questo numero potrebbe essere abbassato drasticamente in quanto almeno uno degli eventuali divisori di n deve essere al più $\sqrt{n}$ (in caso contrario il prodotto risultante sarebbe certamente maggiore di n). Quindi, n è primo se e solo se non ha divisori compresi tra 2 e $\sqrt{n}$. Quindi possiamo migliorare il nostro algoritmo riducendo notevolmente il numero di operazioni eseguite nel caso peggiore.

Stiamo toccando un altro importante aspetto dell'informatica ovvero l'efficienza degli algoritmi: visto che più sono le operazioni eseguite maggiore è il tempo di esecuzione dell'algoritmo, si richiede che gli algoritmi, oltre ad esser corretti, eseguano in minor numero di operazioni, ovvero siano efficienti.

Ecco l'algoritmo per la verifica della primalità di un intero.

1. Parti da $d = 2$;
2. Se $d \leq \sqrt{n}$

   Se d divide n concludi che n non è primo e termina l'esecuzione;

   Altrimenti incrementa d di uno e torna al passo 2;
3. Concludi che n è primo.

Nel passo 2 dell'algoritmo descritto sopra si verifica se d divide n. Se così è l'algoritmo termina concludendo che n non è primo altrimenti si esegue lo stesso test per l'intero successivo a d. Se si arriva al passo 3 allora nel passo 2 non è stato trovato nessun divisore di n quindi si conclude che n è primo. Infine si osservi come il flusso delle istruzioni eseguite varia in base alla valutazione di condizioni. Ad esempio il passo 3 sarà eseguito soltanto quando $d > \sqrt{n}$; inoltre se si trovasse un d che divide n il passo 3 non verrebbe mai eseguito.

L'obbiettivo finale è far eseguire gli algoritmi ad un calcolatore in maniera automatica. Affinché questo sia possibile bisogna anzitutto stabilire un modello che descrive un esecutore meccanico o calcolatore. Successivamente l'algoritmo deve essere opportunamente codificato in modo che l'esecutore automatico sia in grado di comprenderlo ed eseguirlo. La codifica dell'algoritmo prende il nome di *programma*.

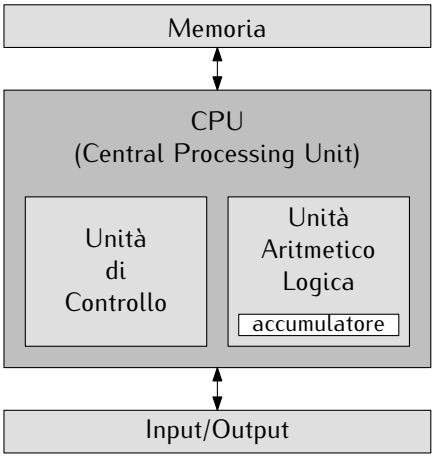


Figura 1.1: La macchina di Von Neumann

1.1 La macchina di Von Neumann

La macchina di Von Neumann è un modello astratto di calcolatore ideato dal matematico John Von Neumann negli anni '40 e poi implementato negli anni successivi. Lo schema della macchina di Von Neumann è illustrato in Figura 1.1. Questa si compone delle seguenti parti.

- La *memoria* che conserva sia il programma (ovvero l'algoritmo opportunamente codificato) che i dati su cui deve lavorare il programma. Può essere vista come un vettore di celle ognuna delle quali contiene una istruzione oppure un dato. Ogni cella è identificabile attraverso un indirizzo che non è altro che un indice del vettore.
- La *CPU* ovvero l'*unità di elaborazione* composta da tre elementi principali:
  - L'*unità aritmetico logica* (*ALU*) esegue le istruzioni elementari come quelle aritmetiche e logiche;
  - L'*unità di controllo* recupera le istruzioni in memoria secondo l'ordine logico stabilito dall'algoritmo e permette la loro esecuzione;
  - L'*accumulatore* è una memoria interna della CPU che viene utilizzata per contenere gli operandi delle istruzioni eseguite dalla ALU.
- L'*input*/*output* (I/O) costituisce l'interfacciamento del calcolatore verso l'esterno (tastiera, dischi, video...).
- Il *bus di comunicazione* è il canale che permette la comunicazione tra le unità appena descritte.

Facciamo riferimento all'algoritmo per la verifica di primalità descritto all'inizio di questo capitolo. Supponiamo che questo sia stato codificato con una serie di istruzioni che possono essere elaborato dalla nostra macchina. Le istruzioni del programma fanno riferimento a dei dati il cui valore regola il susseguirsi delle istruzioni e il risultato finale dell'algoritmo. Questi dati sono il valore di n che rappresenta anche il dato di ingresso o *input* del problema ed il valore di d che viene usato come dato di appoggio utile nel test nel passo 2. Questi due valori sono memorizzati in due celle di memoria in un'area riservata ai dati che è distinta dall'area riservata alle istruzioni del programma. Solitamente ci si riferisce a questi dati utilizzando dei nomi simbolici come appunto n e d chiamati *variabili*. In definitiva possiamo vedere i nomi delle variabili n e d come indirizzi di memoria generici, quando questi nomi vengono invocati all'interno del programma ci si riferisce al valore contenuto nella cella di memoria a cui questi si riferiscono.

Per vedere come funziona la macchina di Von Neumann riscriviamo l'algoritmo del test di primalità in una forma più vicina ad una codifica per la nostra macchina.

ind.	istruzione
1	assegna a d il valore 2
2	se $d - \sqrt{n} \leq 0$ vai a 3, altrimenti vai a 6
3	se $n \equiv 0 \bmod d$ vai a 8
4	incrementa d di 1
5	vai a 2
6	il numero è primo
7	fine
8	il numero non è primo
9	fine

Ogni riga rappresenta una istruzione memorizzata in una cella di memoria il cui indirizzo è specificato da un intero[1].

Nel momento in cui il programma viene eseguito, l'unità di controllo preleva la prima istruzione del programma (quella che si trova nell'indirizzo 1) che verrà eseguita dalla CPU. Nel nostro caso questa istruzione assegna un valore ad una variabile, quindi la CPU scriverà il valore 2 nella locazione di memoria che indicheremo con il nome *d*. Normalmente l'unità di controllo esegue l'istruzione successiva all'ultima già eseguita. In questo caso viene eseguita l'istruzione nell'indirizzo 2. Questa è una *istruzione di controllo* che permette di alterare il flusso del programma in funzione del verificarsi o meno di una condizione. In particolare, se il valore contenuto in *d* è minore o uguale alla radice quadrata del valore contenuto in *n* (operazioni eseguite dall'unità aritmetico logica all'interno della CPU) allora l'unità di controllo eseguirà l'istruzione in posizione 3 altrimenti quella in posizione 6. In posizione 3 troviamo un'altra istruzione di controllo: se *d* divide *n* allora viene eseguita l'istruzione in 8 e poi in 9 che causa l'uscita dal programma con esito negativo (*n* non è primo). Altrimenti si eseguono le istruzioni successive (4 e 5) che incrementano *d* di uno e portano il controllo all'istruzione 2.

1.2 Dal linguaggio macchina all'assembly

La memoria di un calcolatore è un vettore di celle contenenti informazioni codificate in qualche modo. Poiché vengono utilizzati circuiti integrati l'unica informazione possibile è data dallo stato elettrico dei componenti (semplificando, c'è corrente – non c'è corrente). Questo tipo di informazione può essere rappresentata in astratto usando due simboli 0 e 1 per i due stati: ogni cella di memoria contiene una sequenza fissata di questi valori denominati *bit*. Questi due simboli rappresenteranno l'alfabeto del calcolatore per mezzo del quale esprimere i programmi e i dati (*sistema binario*). Col termine *byte* è indicata una sequenza di 8 bit. Un byte rappresenta il taglio minimo di informazione che un calcolatore è in grado di gestire. Quindi una cella di memoria contiene un numero intero di byte (per i modelli attuali di personal computer questi sono 4 o 8). Una cella di memoria è anche indicata come *parola* o *word*.

Il *linguaggio macchina* descrive come sono rappresentate le istruzioni e i dati che costituiscono i programmi che possono essere eseguiti direttamente dalla CPU. Supponiamo per semplicità che ogni istruzione occupi soltanto una cella di memoria (in generale possono essere più grandi). Ogni istruzione sarà composta di due parti principali: una parte iniziale chiamata *codice operativo* che indica il tipo di azione che si deve eseguire ed una seconda parte costituita dagli operandi. Per esempio l'istruzione

`se` *valore* ≤ 0 `vai a` *indirizzo 1* `altrimenti vai a` *indirizzo 2*

può avere la seguente codifica

100110	*cond*	*ind1*	*ind2*

dove i primi 6 bit rappresentano il codice operativo, i restanti bit rappresentano gli operandi che in questo caso sono tre anche questi espressi in binario. Si osservi che ogni operando deve occupare un numero predeterminato di bit affinché sia possibile distinguere ogni singolo operando.

[1] In una architettura reale una singola istruzione potrebbe occupare più di un indirizzo di memoria così come un dato. Il numero di celle occupate dipenderà sia dalla capacità di una singola cella che dal tipo di istruzione o tipo di dato.

Nel linguaggio macchina puro gli indirizzi utilizzati come operandi nelle istruzioni (e che si riferiscono a dati o posizioni all'interno del programma stesso) devono essere indirizzi veri. Quindi il programma cambia a seconda della posizione a partire dalla quale questo viene memorizzato. Tuttavia, al momento della progettazione del programma è più comodo utilizzare delle etichette per indicare sia gli indirizzi di memoria utilizzati per i dati che quelli all'interno del programma. Inoltre anche i codici operativi delle istruzioni "sulla carta" possono essere rappresentati da dei codici più semplici da ricordare. Queste rappresentazioni mnemoniche possono essere formalizzate così da costituire un'astrazione del linguaggio macchina chiamato *linguaggio assembly*. Soltanto al momento della effettiva memorizzazione del programma in memoria per la sua esecuzione sarà necessario sostituire i codici mnemonici e le etichette con i codici operativi e gli indirizzi reali. Questa operazione viene eseguita meccanicamente da un programma chiamato *assembler*.

1.2.1 Codificare i dati

Sia che il programma venga scritto utilizzando direttamente il codice macchina, sia che venga scritto in assembly e poi tradotto con un assembler, sia – come vedremo più avanti – che venga scritto utilizzando un più comodo *linguaggio ad alto livello* il risultato è lo stesso: una sequenza di 0 e 1 che definisce sia i programmi che i dati. In questa sezione descriveremo brevemente come vengono codificati i dati.

Gli interi Siamo abituati a rappresentare i numeri interi positivi utilizzando il *sistema decimale*, ovvero attraverso una sequenza di cifre da 0 a 9. Il valore rappresentato da una cifra all'interno della sequenza dipende dalla posizione che assume la cifra nella sequenza stessa. Per esempio si consideri il numero 1972, la cifra 9 vale 900 ovvero 9×10^2 dove 2 è proprio la posizione della cifra 9 all'interno della sequenza (si assume che la cifra più a destra sia in posizione 0). Quindi

$$1972 = 1 \times 10^3 + 9 \times 10^2 + 7 \times 10^1 + 2 \times 10^0.$$

In definitiva nel sistema decimale il valore dalla cifra in posizione k equivale al valore associato alla cifra moltiplicato per 10 elevato alla k. L'intero 10 ovvero il numero di cifre di cui disponiamo rappresenta la base del sistema. Tuttavia questo numero non ha nulla di magico e quindi può essere sostituito con qualsiasi altra base.

Nel sistema binario abbiamo a disposizione solo due cifre quindi la cifra in posizione k dalla destra ha un valore che vale $b \times 2^k$, dove b vale 0 o 1. Ad esempio

$$\begin{aligned} 11110110100 &= 1 \times 2^{10} + 1 \times 2^9 + 1 \times 2^8 + 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + \\ &\quad 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 1024 + 512 + 256 + 128 + 32 + 16 + 4 = 1972. \end{aligned}$$

Quanto appena detto ci fornisce un metodo per convertire numeri binari in numeri decimali, ma come avviene il processo inverso?

D'ora in poi adotteremo la convenzione di indicare la base di un numero come pedice del numero stesso quando questa non sarà chiara dal contesto. Ad esempio dalla precedente conversione deduciamo che $11110110100_2 \equiv 1972_{10}$. Ora il nostro problema sarà il seguente: dato un intero N in base 10, qual è la sua rappresentazioni in base 2, ovvero qual è la sequenza binaria B tale che $N_{10} \equiv B_2$? Supponiamo di conoscere la rappresentazione binaria di $\lfloor N/2 \rfloor$, sia questa $B' = b'_k b'_{k-1} \ldots b'_0$ allora se b_0 è il resto della divisione di

N per 2 (ovvero $b_0 \in \{0, 1\}$) si ha

$$\begin{aligned} N &= 2\lfloor N/2 \rfloor + b_0 \\ &= 2\left(b'_k \times 2^k + \ldots + b'_0 \times 2^0\right) + b_0 \\ &= b'_k \times 2^{k+1} + \ldots + b'_0 \times 2^1 + b_0 \times 2^0. \end{aligned}$$

Quindi $N_{10} = (b'_k b'_{k-1} \ldots b'_0 b_0)_2$ ovvero la rappresentazione binaria di N è data dalla rappresentazione binaria di $\lfloor N/2 \rfloor$ seguita - a destra - dal bit che rappresenta il resto della divisione di N per 2. Queste osservazioni inducono un algoritmo per la conversione di qualunque intero N nella sequenza binaria corrispondente.

1. Sia $k = 0$
2. Se $N \geq 1$

   Sia b_k il resto della divisione di N per 2;
   Sia $N = \lfloor N/2 \rfloor$;
   Sia $k = k + 1$;
3. Restituisci $b_k b_{k-1} \ldots b_0$

Nella tabella che segue troviamo i valori di N, $\lfloor N/2 \rfloor$ e b_k ottenuti con l'algoritmo se $N = 13$.

N	$\lfloor N/2 \rfloor$	b_k
13	6	$b_0 = 1$
6	3	$b_1 = 0$
3	1	$b_2 = 1$
1	0	$b_3 = 1$

Quindi $13_{10} = 1101_2$.

La codifica degli interi negativi può essere fatta in diversi modi, quello più semplice prevede di utilizzare uno dei bit che si ha a disposizione come *bit di segno*. Per esempio se il bit più significativo di una sequenza è lo 0 allora il numero che segue è positivo altrimenti è negativo. Altri modi per rappresentare interi negativi sono la rappresentazione *complemento a 2* e la rappresentazione *in eccesso* (si veda l'Appendice D di [1]).

I razionali Analogamente a quanto avviene per la rappresentazione in base 10, la cifra b_{-i} in posizione i a destra della virgola vale $b \times 2^{-i}$. I numeri razionali possono essere rappresentati in modo molto semplice utilizzando la notazione a *virgola fissa*. Ovvero, se si hanno a disposizione n bit per rappresentare tali numeri, una parte di essi (diciamo k meno significativi) vengono utilizzati per la parte razionale mentre i restanti $n-k$ vengono utilizzati per la parte intera. Se ad esempio $n = 8$ e $k = 2$ la sequenza 10010101 rappresenta il numero 100101.01_2 che corrisponde a $37 + 2^{-2} = 37.25_{10}$. Questa rappresentazione ha lo svantaggio di "sprecare" bit nel caso in cui, per esempio, si rappresentano interi oppure nel caso in cui i numeri da rappresentare sono molto piccoli (tra 0 e 1). Questo problema viene risolto usando una codifica della rappresentazione scientifica del numero, ovvero il numero x viene scritto come $m \times 2^e$ e le singole parti vengono codificate usando interi. Si ottiene una rappresentazione a *virgola mobile* del numero. Lo standard adottato (IEEE 754) rappresenta un numero in virgola mobile utilizzando di 32 bit. Il primo di questi è il bit di segno; i successivi 8 vengono utilizzati per l'esponente e[2], i restanti per rappresentare la mantissa m[3].

[2] L'esponente può essere negativo: la rappresentazione utilizzate è quella in eccesso

[3] La mantissa è della forma $1.y$, si rappresenta solo la parte dopo la virgola

I caratteri ed il codice ASCII I caratteri (lettere, cifre, punteggiatura, spaziatura) vengono rappresentati attraverso un sistema di codifica a 7 bit denominato *codice ASCII*[4] (American Standard Code for Information Interchange). Poiché con 7 bit si possono ottenere 2^7 diverse sequenza binarie, ne consegue che i caratteri rappresentabili col codice ASCII sono 128. Inoltre ogni sequenza binaria del codice ASCII può essere vista come un numero intero da 0 a 127 questo implica che esiste una corrispondenza uno-a-uno tra gli interi da 0 a 127 ed i caratteri. Nella tabella che segue sono mostrati i codici ASCII dei caratteri stampabili, i caratteri mancanti sono spazi, tabulazioni, ritorno-carrello e altri caratteri di controllo. I codici sono rappresentati in forma decimale.

Cod.	Car	Cod.	Car	Cod.	Car	Cod.	Car	Cod.	Car
33	!	52	4	71	G	90	Z	109	m
34	"	53	5	72	H	91	[	110	n
35	#	54	6	73	I	92	\	111	o
36	$	55	7	74	J	93	]	112	p
37	%	56	8	75	K	94	^	113	q
38	&	57	9	76	L	95	_	114	r
39	'	58	:	77	M	96	`	115	s
40	(	59	;	78	N	97	a	116	t
41	)	60	<	79	O	98	b	117	u
42	*	61	=	80	P	99	c	118	v
43	+	62	>	81	Q	100	d	119	w
44	,	63	?	82	R	101	e	120	x
45	-	64	@	83	S	102	f	121	y
46	.	65	A	84	T	103	g	122	z
47	/	66	B	85	U	104	h	123	{
48	0	67	C	86	V	105	i	124	|
49	1	68	D	87	W	106	j	125	}
50	2	69	E	88	X	107	k	126	~
51	3	70	F	89	Y	108	l		

Si osservi che i caratteri alfabetici minuscoli hanno codici consecutivi (il carattere 'a' ha codice 97, 'b' ha codice 98 e via dicendo). La stessa cosa vale per i caratteri maiuscoli e per i caratteri rappresentanti i numeri da 0 a 9.

1.3 I linguaggi ad alto livello

Il linguaggio macchina ed il linguaggio assembly presentano diversi inconvenienti. Sono dipendenti dall'architettura: ogni processore ha un suo insieme di istruzioni specifico quindi il codice scritto non è portabile. I programmi sono di difficile lettura quindi non adatti alla descrizione di algoritmi complessi.

Per semplificare il compito ai programmatori sono stati introdotti i linguaggi di programmazione ad *alto livello*. Un linguaggio di programmazione ad alto livello è definito in modo da essere svincolato dall'architettura del calcolatore; inoltre può utilizzare elementi del linguaggio naturale rendendo il linguaggio più semplice da utilizzare. Infine, grazie alla sua astrazione da una specifica architettura, i suoi programmi possono essere facilmente trasportati da una macchina ad un altra. Tuttavia il linguaggio deve avere regole precise

[4] pronuncia 'aschi' o 'asci'

e non ambigue così che i programmi possano essere tradotti automaticamente in linguaggio macchina e quindi essere eseguiti da un calcolatore.

Il processo di traduzione automatica da un programma scritto in un linguaggio ad alto livello al linguaggio macchina è detto *compilazione* ed il programma che esegue il processo è detto *compilatore*. È questo il processo che instanzia un programma astratto scritto con un linguaggio ad alto livello su una specifica architettura. Attraverso questo processo siamo in grado di far eseguire un programma P scritto nel linguaggio ad alto livello L su le architetture $A_1, A_2, \ldots, A_k$ tutte diverse tra di loro. Per far questo è sufficiente avere un compilatore C_i che trasforma programmi scritti nel linguaggio L in codice macchina per l'architettura A_i.

$$P \rightarrow C_i \rightarrow P_i$$

Ovvero il compilatore C_i prende in input il programma P scritto nel linguaggio L e produce in output il programma P_i scritto in codice macchina per l'architettura A_i. Ovviamente i programmi P e P_i devo essere equivalenti!

Nel prossimo capitolo descriveremo il linguaggio C che utilizzeremo in queste note.

Capitolo 2

Il linguaggio C, un primo approccio

Il linguaggio C è stato introdotto negli anni '70 e fu pensato per scrivere sistemi operativi. Si è imposto negli anni a venire grazie alla sua efficienza e duttilità in tutti gli ambiti applicativi.

Prima di passare alla descrizione del linguaggio diamo prima un esempio di un programma minimale in C che calcola l'area di un triangolo di base 3 ad altezza 5

```
1 main(){
2   float area;
3   int base, altezza;
4
5   base = 3;
6   altezza = 5;
7
8   area = base*altezza/2.0;
9 }
```

Un programma C deve contenere una (ed una sola) funzione *main*. Le istruzioni contenute in essa saranno le prime ad essere eseguite al momento in cui si lancia il programma. Questo può contenere altre funzioni che saranno invocate a partire da qualche istruzione nella funzione main. Le istruzioni sono separate tra di loro da un punto e virgola, inoltre più istruzioni possono essere raggruppate in un *blocco* delimitato dalle parentesi graffe aperta e chiusa. La funzione main contiene una singola istruzione oppure, come nell'esempio, un blocco di istruzioni. Il programma in questione presenta due tipi di istruzioni: le prime due nelle righe 1 e 2 sono *dichiarazioni di variabili*.

2.1 Variabili

Una *variabile* è un nome simbolico associato ad uno spazio di memoria che conterrà dei dati. In base al tipo di dato la zona di memoria può essere più o meno grande ed è per questo che, al momento in cui si definisce la variabile, bisogna specificare di che tipo di dato si tratta. In particolare l'istruzione

```
float area;
```

è una *dichiarazione* di variabile. Ha l'effetto di creare una variabile di nome `area` a cui associa uno spazio di memoria sufficiente a contenere un numero in virgola mobile; si dice che viene creata una variabile, `area`, di tipo `float`. Mentre l'istruzione in riga 3

```
int base, altezza;
```

crea due variabili variabili di tipo `int`, ovvero `base` e `altezza` faranno riferimento a spazi di memoria di dimensioni tali da contenere numeri interi. In generale lo spazio riservato ad una variabile `float` può essere diverso da quello riservato ad una variabile `int`.

Le due istruzioni che seguono (linea 5 e linea 6) sono due istruzioni di *assegnamento*, ovvero alle variabili `base` e `altezza` vengono assegnati i valori 3 e 5 rispettivamente. L'effetto è che questi valori numerici sono scritti nell'area di memoria associata alle due variabili.

Anche l'ultima istruzione contiene un assegnamento, questa volta alla variabile `float area`. Il valore assegnato è dato dal risultato di una operazione aritmetica che prevede una moltiplicazione tra i due valori associati alle due variabili `base` e `altezza` seguita da una divisione per una costante (2.0). Nella tabella che segue sono elencati i tipi di dato in C con il numero di bit che solitamente occupano nelle implementazioni più comuni.

Tipo	Bit	Descrizione
`char`	8	Intero
`short`	16	Intero
`int`	32	Intero
`long`	32	Intero
`long long int`	64	Intero
`float`	32	Virgola mobile
`double`	64	Virgola mobile
`long double`	96	Virgola mobile

Si tenga conto che il numero di bit assegnato ai tipi può differire in base all'architettura.

Il C, a differenza di altri linguaggi più "tipati", permette di utilizzare nelle istruzioni elementi di tipo diverso. Per esempio è possibile sommare un `float` ad un `char` e assegnare il risultato ad un `int`. Tuttavia bisogna aver presente due cose: il tipo risultante di una operazione è dato dal tipo più grande (in termini di byte) tra tutti quelli coinvolti (il tipo risultate tra la somma di un `float` ed un `char` è un `float`); l'assegnazione di un'espressione ad una variabile di tipo più piccolo rispetto a quello dell'espressione potrebbe comportare la perdita di dati in quanto la variabile destinataria ha una capacità minore, il viceversa non produce effetti collaterali (si può assegnare ad un `int` il risultato della somma tra due `float` ma si rischia di perdere informazioni).

Con i tipi interi si possono rappresentare 2^k valori dove k è il numero di bit relativo al tipo specifico. Questi 2^k interi sono sia positivi che negativi, per distinguere i positivi dai negativi si utilizza un bit di segno, quindi gli interi che si possono rappresentare variano nell'intervallo $-2^{k-1}-1, \ldots, 2^{k-1}-1$ (in tutto i numeri rappresentati sono 2^k-1 e non 2^k in quanto lo zero compare con entrambi i segni). Facendo precedere la dichiarazione di variabile intera dal modificatore `unsigned` il bit di segno avrà lo stesso scopo degli altri bit quindi si potranno rappresentare tutti gli interi tra 0 e 2^k-1. Ad esempio poiché le lunghezze sono grandezze non negative, nel programma precedente avremmo potuto scrivere

```
unsigned int base, altezza;
```

In questo modo, i valori ammissibili per le due variabili variano da 0 a $2^{32}-1$ anziché da -2^{31} a $2^{31}-1$.

2.2 Operatori

Gli *operatori* di un linguaggio di programmazione, ed in particolare del linguaggio C, permettono di manipolare i dati eseguendo su di essi operazioni aritmetiche, logiche e di assegnazione analoghe alle operazioni eseguite dagli operatori matematici. Nel programma di esempio abbiamo già utilizzato diversi operatori. Tra questi gli *operatori aritmetici* di moltiplicazione ($*$) e divisione ($/$). A questi si aggiungono gli operatori di somma ($+$), sottrazione ($-$) e modulo ($\%$) che restituisce il resto della divisione intera tra due numeri.

Altro operatore utilizzato è l'*operatore di assegnamento* $=$ che permette di assegnare ad una variabile uno specifico valore.

```
var = val;
```

Questa istruzione assegna alla variabile `var` il valore `val`. Il C dispone di altri operatori di assegnamento che contengono anche operatori aritmetici. Ad esempio

```
var++; ++var;
var--; --var;
```

Le prime due istruzioni sono di *incremento unario* e altre due sono di *decremento unario* ed equivalgono rispettivamente alle due istruzioni che seguono

```
var = var + 1;
var = var - 1;
```

Ovvero assegnano alla variabile `var` il suo vecchio valore incrementato (o decrementato) di uno. Per saperne di più circa la differenza tra i due operatori di incremento e decremento unario e per informazioni circa gli altri operatori di assegnamento si consulti il Capitolo 9 di [1].

2.3 Controllo del flusso ed operatori logici e relazionali

Le istruzioni del programma illustrato all'inizio di questo capitolo vengono seguite dalla prima all'ultima secondo un ordine sequenziale. Spesso però, nella soluzione dei problemi, è necessario poter disporre di meccanismi che permettano di eseguire delle operazioni al verificarsi di un dato evento. Nell'esempio del test di primalità trattato nel Capitolo 1 si testava se un intero d, inizializzato a 2, divideva o meno il potenziale numero primo n. Se d non lo divideva ed era minore di o uguale di $\sqrt{n}$ veniva eseguito lo stesso test per il numero successivo a d. Se si verificava che d divideva n si concludeva che n non era primo, mentre se si verificava che $d > \sqrt{n}$ si concludeva che n era primo. Per poter scrivere un programma che implementi questo algoritmo in C abbiamo bisogno di un paio di costrutti. Il primo di questi ci deve consentire di eseguire la stessa operazione (incrementare d) fintanto che risulta verificata una determinata condizione ($d \leq \sqrt{n}$ e d non divide n). Il secondo costrutto ci permette di concludere circa la primalità di n quando la condizione precedente non è più verificata. In particolare, se non è verificata perché $d > \sqrt{n}$ allora n è primo, se invece non è verificata perché d divide n allora n non è primo.

In C scriveremmo quanto segue

```
#include <stdio.h>
#include <math.h>

main(){
    int n = 691;
    int d = 2;
    while( (d <= sqrt(n)) && (n % d != 0)){
        d++;
    }
    if( d > sqrt(n) )
        printf("Il numero e' primo\n");
    else
        printf("In numero non e' primo\n");
}
```

Questo programma utilizza il costrutto `while` la cui forma è la seguente:

```
while(condizione)
    istruzione;
```

dove `istruzione` può essere anche un blocco di istruzioni delimitato da parentesi graffe, questa convenzione verrà utilizzata anche in seguito. L'effetto di questo costrutto è quello di eseguire `istruzione` fintanto che `condizione` risulta essere verificata.

Nel nostro programma la condizione del `while` è la congiunzione (&&) di due sotto-condizioni $d \leq \sqrt{n}$ e d non divide n. La seconda sotto-condizione è espressa utilizzando l'operatore modulo (%), ovvero il resto della divisione di n per d è diverso (! =) da zero.

Il secondo costrutto utilizzato ci permette di trarre le opportune conseguenze in base al perché il programma è uscito dal `while`. Si esce dal ciclo indotto dal `while` perché una delle due sotto-condizioni non è più verificata. Con l'istruzione `if-else` stabiliamo quale delle due sotto-condizioni non è più verificata. La forma dell'istruzione `if-else` è la seguente:

```
if(condizione)
    istruzione1;
else
    istruzione2;
```

Questo costrutto crea una diramazione del flusso del programma che prende una via o l'altra in base al valore di `condizione`. In particolare se `condizione` è vero viene eseguito il blocco o istruzione singola `istruzione1` altrimenti viene eseguito `istruzione2`.

Nel caso del nostro test di primalità viene stampato a video il messaggio "Il numero e' primo se $d > \sqrt{n}$ altrimenti viene stampato il messaggio "Il numero non e' primo".

L'output del messaggio è eseguito con l'istruzione `printf` che ha l'effetto di stampare a video la stringa in input racchiusa tra doppi apici ("). Entrambe le stringhe terminano con \n che serve ad aggiungere un fine riga al messaggio. Nei prossimi capitoli incontreremo altri utilizzi delle funzioni di input/output ma se si vuole avere una panoramica completa sulle questioni riguardanti l'input e l'output in C si consulti il Capitolo 7 di [4].

Una ultima osservazione riguarda le prime due righe del nostro programma

```
#include <stdio.h>
#include <math.h>
```

Queste sono necessarie se si intende utilizzare funzioni di input-output come `printf` oppure funzioni matematiche come **sqrt**. Su questo argomento torneremo alla fine del capitolo.

Operatori logici Abbiamo visto come il comportamento delle istruzioni `while` e `if-else` sia determinato dalla valutazione di una condizione di verità. Nel linguaggio C il valore di una condizione è un numero intero. Questo è interpretato come vero se è diverso da zero mentre è interpretato come falso se è uguale a zero. Quindi il seguente codice è legale

```
while (1)
    istruzione
```

La condizione nel `while` è un intero diverso da zero, quindi è sempre verificata. L'effetto di questo frammento di programma è la ripetizione all'infinito di `istruzione`.

Più condizioni possono essere relazionate tra di loro per mezzo di *operatori logici*. Uno di questo lo abbiamo già visto, era l'operatore *and* o congiunzione che in C viene espresso con &&. Intuitivamente l'and di due condizioni è vero se entrambe le condizioni sono vere, falso altrimenti. Altri operatori logici sono l'*or*, il *not*. L'or di due condizioni è vero se almeno una delle due condizioni è vera, falso altrimenti. In C l'or si estrime con ||. L'and e l'or sono operatori binari perché agiscono su due argomenti. Invece il *not* è un operatore unario: esso agisce su un unica condizione invertendone il valore. In C si indica con !.

Assumendo la condizione vera sia associato a 1 e falsa a 0, le tabelle che seguono riassumono il comportamento dei tre operatori logici.

c_1	c_2	c_1 && c_2	c_1 || c_2
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

c	!c
0	1
1	0

Operatori relazionali Questi agiscono su due valori numerici e restituiscono 1 o 0 a secondo che la relazione tra i due valori sia vera o falsa. Di seguito sono elencati gli operatori relazionali in C ed il loro comportamento in base ai valori degli argomenti.

Operatore	Risultato
val1 $<$ val2	1 se e solo se val1 $<$ val2
val1 $<=$ val2	1 se e solo se val1 $\leq$ val2
val1 $>$ val2	1 se e solo se val1 $>$ val2
val1 $>=$ val2	1 se e solo se val1 $\geq$ val2
val1 $==$ val2	1 se e solo se val1 $=$ val2
val1 $!=$ val2	1 se e solo se val1 $\neq$ val2

Poiché il risultato di un operatore relazionale è uguale a 0 se la condizione è falsa e 1 altrimenti, questo può essere utilizzato come argomento delle istruzioni di controllo del flusso come abbiamo già fatto nel test di primalità.

2.4 Gli array

Supponiamo di voler scrivere un programma che faccia delle statistiche sulle temperature del mese di agosto. Ad ogni giorno del mese associamo un numero razionale che descrive la temperatura massima di quel giorno. La prima cosa che potremmo fare è calcolare il valore più alto e quello più basso memorizzato. Come memorizzare le 31 temperature richieste? Potremmo usare 31 variabili, ma la cosa risulterebbe alquanto scomoda. Ci occorrerebbe un modo più comodo di memorizzare e organizzare i nostri dati in modo che possano essere manipolati efficientemente. Ovvero una *struttura dati* opportuna.

Quello che ci occorre è un modo di memorizzare una collezione di 31 valori, tutti dello stesso tipo in un'unica struttura. Supponiamo che la temperatura sia espressa con un `float`, allora potremmo risolvere in questo modo

```
float temperatureAgosto[31];
```

Abbiamo definito un vettore di 31 elementi ognuno dei quali è un `float`. Ad ogni elemento è associato un indice che va da 0 (primo elemento del vettore) a 30 (ultimo elemento del vettore) che è utilizzato per accedere all'elemento specifico. Per esempio se vogliamo impostare la temperatura massima del primo e del quindicesimo giorno del mese scriveremo

```
temperatureAgosto[0] = 32.0;
temperatureAgosto[14] = 34.0;
```

Con questo esempio abbiamo creato e manipolato un array ad una dimensione.

Un *array* è una struttura dati del C (e non solo) che si rifà al concetto matematico di vettore o matrice. Permette di memorizzare una collezione di elementi tutti dello stesso tipo. Ogni elemento è individuato all'interno dell'array attraverso uno o più indici interi. Ogni array ha un numero fissato di dimensioni ed ogni dimensione ha una lunghezza anche questa fissata al momento della creazione dell'array.

La creazione di un array ad una dimensione di N elementi di un tipo `type` avviene in questo modo

```
type nomearray[N];
```

dove N deve essere un valore numerico intero. Se vogliamo riferirci all'elemento in posizione i dell'array con $0 \leq i < N$ useremo la sintassi

```
nomearray[i];
```

Vediamo un esempio di utilizzo: vogliamo scrivere un programma che calcola il valore massimo delle temperature del mese di agosto.

```
#include <stdio.h>

main(){
    float temperatureAgosto[] = { 32.0, 33.0, 31.0,
        28.0, 32.0, 33.0, 34.0, 33.0, 32.0, 31.0,
        34.0, 33.0, 33.0, 33.0, 34.0, 35.0, 36.0,
        36.0, 37.0, 36.0, 36.0, 34.0, 36.0, 34.0,
        33.0, 34.0, 34.0, 34.0, 32.0, 32.0, 33.0};
    float max = 0;
    int i = 0;
    while( i<31 ){
        if ( temperatureAgosto[i] > max )
            max = temperatureAgosto[i];
        i++;
    }
    printf("Temperatura massima = %f\n", max);
}
```

In questo programma viene creato un vettore di 31 elementi `float` in modo esplicito ovvero si elencano tutti gli elementi del vettore consecutivamente separati da virgola all'interno di una coppia di parentesi graffe, si noti che la dimensione del vettore non viene esplicitata in quanto viene inferita dal numero di elementi elencati. Segue l'analisi dei dati in cui si confronta ogni elemento del vettore con la variabile `max` che contiene il valore massimo provvisorio. Il valore di `max` viene sostituito col valore dell'elemento del vettore analizzato quando quest'ultimo risulta maggiore di `max`.

Nell'esempio precedente come anche in quelli che lo precedevano, i dati su cui opera il programma sono contenuti all'interno del codice stesso del programma: gli elementi del vettore sono stabiliti al momento in cui il programma viene progettato. Per quanto vedremo in seguito questa pratica è poco comoda. Quindi è buona norma slegare i programmi dai dati su cui questi agisco, per esempio il programma potrebbe essere progettato in modo che i dati vengano forniti dall'utente attraverso la tastiera. Il programma che segue è una evoluzione del precedente in cui le temperature vengono fornite dall'utente attraverso la tastiera durante l'esecuzione del programma.

```
#include <stdio.h>

main(){
    float temperatureAgosto[31];
    float max = 0;
    int i = 0;

    while( i<31 ){
        printf("Temperatura del giorno %d ",i);
        scanf("%f", &(temperatureAgosto[i]));
        i++;
    }

    i=0;
    while( i<31 ){
        if ( temperatureAgosto[i] > max )
            max = temperatureAgosto[i];
        i++;
    }
    printf("Temperatura massima = %f\n", max);
}
```

Con il primo ciclo `while` avviene l'acquisizione dell'input, ovvero delle temperature. Ad ogni passo all'interno del ciclo viene eseguita l'istruzione `scanf` che ha l'effetto di interrompere il programma mettendosi in attesa dell'input dell'utente che arriverà attraverso la tastiera. Questo sarà composto da una serie di caratteri seguiti da un *invio*. La `scanf` prende la sequenza di caratteri, prova a convertirla in un numero `float` (come specificato dal parametro `%f` nel primo argomento della funzione) e, nel caso di successo, il valore ottenuto lo assegna alla variabile `temperatureAgosto[i]`. Si osservi che la variabile in questione è preceduta dall'operatore `&`. Il motivo di questo sta nel fatto che la funzione `scanf` scrive un valore in una locazione di memoria individuata da una variabile. Col nome della variabile intendiamo il suo valore e non la sua locazione di memoria, questa viene ottenuta anteponendo l'operatore `&` al nome della variabile.

2.4.1 Stringhe

Le stringhe sono sequenze di caratteri. In C, non essendoci un tipo specifico per le stringhe, queste vengono rappresentate come vettori di caratteri, ovvero vettori di tipo `char`.

Si ricorda che il `char` è un intero solitamente solitamente composto da 8 bit (un byte), che quindi può essere utilizzato per rappresentare $2^8 = 256$ diversi interi sufficienti per coprire tutti i codici ASCII (si veda il Capitolo 1).

```
char c = 97;
```

L'istruzione precedente crea una variabile `c` di tipo `char` alla quale viene assegnato il valore 97 che è il codice ASCII del carattere `a`. Lo stesso effetto si poteva ottenere con la seguente istruzione

```
char c = 'a';
```

Con gli apici '_' si accede al codice ASCII corrispondente al carattere che questi racchiudono.

Come si è detto, in C le stringhe vengono rappresentate con vettori di `char`.

```
char s[100];
```

In questo esempio si è definita una stringa che può contenere al più 100 caratteri. Può contenerne anche di più piccole infatti si adotta la convenzione che l'ultimo carattere della stringa sia il carattere speciale '\0', chiamato carattere di *fine stringa*. Le funzioni che gestiscono le stringhe devono tener conto di questa convenzione, quindi devono ignorare tutti i caratteri che seguono la prima occorrenza di questo all'interno del vettore.

Una stringa può essere definita ed inizializzata come un vettore. Ecco alcuni esempi.

```
char s[100];
s[0] = 'c';
s[1] = 'i';
s[2] = 'a';
s[3] = 'o';
s[4] = '\0';
```

```
char s[100] = {'c', 'i', 'a', 'o', '\0'};
```

Oppure più comodamente elencando i caratteri tra virgolette, senza separatore e senza carattere di fine stringa.

```
char s[100] = "ciao";
```

In questo caso, provvederà il compilatore ad inserire il carattere di fine stringa in posizione opportuna.

È possibile fare in modo che la stringa venga acquisita tramite tastiera. Anche per le stringhe la funzione utilizzata è la `scanf`.

```
1  #include <stdio.h>
2
3  main(){
4    char stringa[100];
5    int numchars;
6    scanf("%s", stringa);
7
8    numchars = 0;
9    while(stringa[numchars]!='\0')
10     numchars++;
11   printf("Numero caratteri = %d\n",numchars);
12 }
```

Nel programma precedente viene acquisita una stringa da tastiera e memorizzata nella variabile `stringa`, poi vengono contati i caratteri che la compongono e, finalmente, viene stampato un rapporto in output. L'acquisizione avviene con la funzione `scanf`: ora i caratteri

in input devono essere convertiti in una stringa (come indicato dal parametro `%s`[1] nel primo parametro della funzione), il risultato della conversione va memorizzato nella variabile `stringa`. Si osservi che in questo caso la variabile non viene preceduta dall'operatore &. Questo perché, come è stato detto, la funzione `scanf` vuole come parametro una locazione di memoria in cui memorizzare la quantità convertita. Poiché una variabile stringa è un vettore e il valore di una variabile vettore non altro che la locazione di memoria a partire dalla quale sono memorizzati i dati del vettore, la locazione di memoria da passare alla `scanf` è proprio il contenuto della variabile. Dopo aver acquisito la stringa di input passiamo al conteggio dei caratteri. Questo avviene scandendo il vettore contenente la stringa dal primo elemento fino a quello contenente il carattere di fine stringa.

2.5 Le `struct`

I vettori possono essere visti come dei contenitori di un dato numero di elementi dello stesso tipo. Supponiamo invece di dover memorizzare dati relativi ai correntisti di una banca. Ad ogni correntista corrisponde un insieme di dati tra di loro disomogenei come il nome ed il cognome identificati da due stringhe, il numero di conto identificato per esempio con un intero, il saldo identificato con un `float` e così via. In C è possibile definire dei contenitori di elementi di tipo diverso utilizzabili per raccogliere in un unico oggetto[2] dati disomogenei come quelli dell'esempio appena citato.

Questi contenitori sono chiamati `struct`, gli elementi che definiscono la `struct` sono detti *campi*. La dichiarazione di una `struct` definisce un nuovo tipo di dati utilizzabile nella definizione di variabili. Quello che segue è una dichiarazione di una struttura per la gestione dei conti corrente.

```
struct contocorrente {
    int numeroconto;
    char nome[20];
    char cognome[20];
    float saldo;
};
```

La definizione di una variabile di tipo `struct contocorrente` avviene utilizzanzo la struttura creata come un nuovo tipo.

```
struct contocorrente cc;
```

Abbiamo creato una nuova variabile `cc` di tipo `struct contocorrente`. Per accedere ai singoli campi della variabile `cc` si utilizza l'operatore `.` (punto).

```
cc.saldo;
```

In questo modo si accede al campo `saldo`, di tipo `float` della variabile `cc`.

[1] Tra gli altri parametri di conversione ci sono %c per i `char`, %d per gli interi. Per una rassegna completa si consiglia la consultazione del Capitolo 7 di [4]

[2] In questo documento il termine "oggetto" va inteso nella sua accezione più generale e non in quella specifica della *programmazione ad oggetti* che non è un argomento preso in considerazione in queste note. Chi non capisce il senso di questa nota stia tranquillo, non ne è il destinatario.

Per semplificare la notazione possiamo definire un nuovo tipo di dati `contocorrente` utilizzato al posto del tipo `struct contocorrente`. Per ottenere questo si utilizza il costrutto `typedef` nel modo che segue.

```
struct contocorrente {
    int numeroconto;
    char nome[20];
    char cognome[20];
    float saldo;
};
typedef struct contocorrente contocorrente;

contocorrente cc;
```

L'istruzione `typedef` appena utilizzata crea un nuovo tipo `contocorrente` identico al tipo già esistente `struct conocorrente`. Ora per definire la variabile `cc` può essere utilizzato il nuovo tipo del tutto equivalente al vecchio `struct contocorrente`.

2.6 Funzioni

Nello scrivere un programma potrebbe succedere di aver bisogno di calcolare il massimo tra due numeri. In C questo problema può essere agevolmente risolto col seguente codice che trova il massimo tra due numeri `x` e `y` e lo assegna ad una terza variabile `m`.

```
if (x > y)
    m = x;
else
    m = y;
```

Se nel nostro programma avessimo bisogno più volte di ricorrere al calcolo del massimo potremmo ripetere il codice sopra illustrato tante volte quante ne occorrono. Tuttavia sarebbe più comodo poter disporre di una funzione o istruzione o primitiva che computi questo valore. Purtroppo il linguaggio C non dispone di tale funzionalità specifica ma mette a disposizione uno strumento più generale che consente di definire nuove funzioni utilizzabili quando se ne ha la necessità. Per esempio potremmo definire una funzione, chiamata `max`, che calcola il massimo tra due interi in input.

```
int max(int a, int b){
    int ris;
    if (a > b)
        ris = a;
    else
        ris = b;
    return ris;
}
```

Ora all'interno del nostro programma, per calcolare il massimo tra le variabili intere `x` e `y` da assegnare alla variabile intera `m` sarà sufficiente scrivere quanto segue.

```
m = max(x, y);
```

Vediamo un esempio completo in cui viene utilizzata la funzione.

```
1  #include <stdio.h>
2
3  int max(int, int);
4
5  main(){
6    int x, y;
7    scanf("%d,%d", &x, &y);
8    printf("massimo tra %d, %d e 0 = %d\n",\
9       x, y, max(max(x,y),0));
10 }
11
12 int max(int a, int b){
13     int m;
14     if (a > b)
15         m = a;
16     else
17         m = b;
18     return m;
19 }
```

Il programma precedente stampa 0 se entrambi gli interi `x` e `y` inseriti dall'utente sono negativi, altrimenti stampa il massimo tra `x` e `y`. Questo viene ottenuto calcolando il massimo tra i tre valori ovvero `x`, `y` e 0. In particolare, all'interno della `printf` viene prima calcolato il massimo tra `x` e `y`. Viene poi stampato il massimo tra il risultato di questa ultima operazione e 0.

Ora andiamo ad analizzare punto per punto le novità nel programma proposto. Nella riga 3 c'è una prima presentazione della funzione `max`, si tratta del *prototipo* della funzione. Serve per definire come deve essere invocata la funzione: quanti sono i suoi argomenti (due), di che tipo sono (entrambi `int`) e che cosa produce in output (un `int`). Qui non si definisce il suo comportamento ma soltanto il modo con cui questa si interfaccia col resto del programma. La definizione vera e propria della funzione inizia a partire dalla riga 12. All'interno della funzione i due parametri assumeranno il nome delle due variabili `a` e `b` che conterranno la copia dei due argomenti utilizzati al momento dell'invocazione della funzione. Dopo aver calcolato il valore massimo tra i due parametri e averlo memorizzato nella variabile `m` viene invocata l'istruzione `return m` che provoca l'uscita dalla funzione ed il ritorno del flusso di esecuzione del programma nel punto in cui è avvenuta l'invocazione della funzione. Al posto dell'invocazione della funzione viene sostituito il valore calcolato dalla funzione stessa. Nel nostro caso nella riga 9 all'interno della `printf` avviene una invocazione della funzione `max` con argomenti il risultato di un'altra invocazione della funzione `max` su `x` e `y` e 0. Ovvero prima verrà calcolato il massimo tra il valore di `x` e `y`, il valore calcolato sarà utilizzato come argomento, insieme allo 0 per un'altra invocazione della funzione. Il risultato finale verrà stampato sullo schermo.

Più volte è stato sottolineato che le variabili utilizzate come parametri nella definizione di una funzione prendono solo il valore degli argomenti passati alla funzione al momento delle sua invocazione. Quindi se all'interno della funzione viene modificato uno dei valori di una

variabile parametro questo avrà solo un effetto locale: la variabile utilizzata all'interno della funzione e quella utilizzata come argomento nella chiamata della funzione sono due cose distinte. Inoltre le variabili di funzione vengono create al momento del lancio della funzione ma poi vengono distrutte all'uscita (dopo l'istruzione `return`). Per quest'ultimo motivo non sono visibili all'esterno. Per fare in modo che le azioni eseguite sulle variabili parametro all'interno di una funzione siano permanenti bisogna passare alla funzione l'indirizzo della variabile da modificare e non il suo valore (come per la `scanf`). Quest'ultimo argomento verrà ripreso in seguito.

Infine ricordiamo che ogni programma C deve contenere almeno la funzione `main`. Al momento dell'esecuzione del programma caricato in memoria viene lanciata per prima questa funzione. Come per le altre funzioni, le variabili dichiarate all'interno di essa non sono visibili all'esterno (e quindi all'interno di altre funzioni).

2.7 Il processo di compilazione

Come si è discusso nel Paragrafo 1.3, il programma scritto nel nostro linguaggio ad alto livello (il C) deve essere tradotto nel linguaggio macchina del nostro calcolatore e caricato in memoria. Questo processo viene gestito da un secondo programma chiamato *compilatore*.

Il programma ad alto livello viene scritto attraverso il nostro editor di testo preferito e salvato in memoria con un nome, per esempio `test_primalita.c`. Per compilarlo possiamo utilizzare il compilatore `gcc` lanciando il programma da terminale specificando come argomento il nome del file contenente il programma da compilare

```
gcc test_primalita.c
```

Se non ci sono errori il risultato della compilazione sarà un file eseguibile che quindi può essere lanciato. Il nome attribuito al file dipende dal sistema operativo che si sta utilizzando. Se si vuole imporre un nume al file risultante questo può essere specificato come ulteriore armomento da passare a `gcc` nel seguente modo:

```
gcc test_primalita.c -o test_primalita
```

In realtà la compilazione invocata dall'istruzione precedente non coinvolge solo il file `test_primalita.c`. Infatti vengono aggiunti un insieme di file che contengono il codice delle funzioni di libreria. Queste sono funzioni esterne, quindi non proprie del linguaggio C che, sotto forma di file pre-compilati, vengono collegati (*link*) al risultato della compilazione del nostro programma. Questi file pre-compilati sono detti *librerie*. Per esempio le funzioni `printf` e `scanf` sono contenute nella libreria `stdio`, la funzione `sqrt` è contenuta nella libreria `math` eccetera. Nel file in cui queste funzioni vengono utilizzate occorre specificarne i prototipi che sono contenuti nei file `.h` che vengono inclusi all'inizio del file (per esempio `stdio.h` e `math.h`). Molte librerie vengono collegate di default dal compilatore (come la `stdio`) mentre altre devono essere collegate esplicitamente. Per esempio la libreria `math` è una di queste. Quindi per compilare correttamente il programma basta aggiungere l'opzione `-lm` come segue

```
gcc test_primalita.c -o test_primalita -lm
```

Capitolo 3

La valutazione degli algoritmi

Consideriamo il problema della disposizione a tavola (rotonda) dei prodi e rissosi cavalieri di Re Artù: i cavalieri in questione, anche se coraggiosi e fedeli al loro re, non sono tutti amici tra loro, ma accade spesso che coppie di cavalieri siano nemici da tenere lontani. Quel che Artù vuole è disporre i suoi n cavalieri intorno alla tavola rotonda (che ha n seggi intorno), in modo tale che due cavalieri seduti vicini non siano nemici.

Prima di affrontare il problema dobbiamo innanzi tutto codificarlo, in particolare dobbiamo decidere come rappresentare i dati e l'eventuale soluzione. I cavalieri sono n e quindi possiamo mettere in relazione uno-a-uno l'insieme dei cavalieri con gli interi da 0 a $n-1$. Per rappresentare i rapporti tra i cavalieri possiamo utilizzare una matrice $n \times n$ chiamata N a valori 0 o 1 avente il seguente significato: $N[x, y] = 1$ se e solo se i cavalieri x e y sono nemici. Chiaramente la matrice N è simmetrica. Una soluzione non è altro che una permutazione dei cavalieri: enumerando i posti intorno alla tavola con interi da 0 a $n-1$ possiamo rappresentare una soluzione con un vettore P di n elementi a valori in $\{0, \ldots, n-1\}$ col seguente significato $P[i] = x$ se in posizione i siede il cavaliere x. Una soluzione P deve verificare la seguente proprietà:

$$\text{per ogni } i \in \{0, \ldots, n-1\} \quad N[P[i], P[i+1 \bmod n]] = 0. \tag{3.1}$$

Avendo la matrice N ed un vettore delle permutazione P è facile costruire una funzione per la verifica della proprietà espressa dall'Equazione 3.1. Prima di costruire la funzione vediamo come rappresentare matrici nel linguaggio C.

Nel linguaggio C la matrice può essere implementata attraverso array bi-dimensionali. Un array bi-dimensionale non è altro che un array di array. Ovvero ogni elemento dell'array contiene le righe della matrice che, a loro volta, sono array.

```
int N[7][7] = {
     {0,1,0,0,1,0,1},
     {1,0,1,0,0,0,0},
     {0,1,0,0,0,0,1},
     {0,0,0,0,0,1,0},
     {1,0,0,0,0,0,0},
     {0,0,0,1,0,0,0},
     {1,0,1,0,0,0,0}
};
```

La definizione della matrice N è una generalizzazione della definizione di un vettore, prima si specifica la dimensione del vettore che contiene le righe della matrice e poi la dimensione di ogni riga. La matrice viene anche inizializzata, quindi vengono elencati uno per uno i vettori che rappresentano le righe dalla matrice.

La funzione che segue restituisce 1 se il vettore P è una permutazione corretta rispetto i vincoli imposti dalla matrice N, restituisce 0 altrimenti. Attenzione, il simpolo n che compare nel codice non è una variabile ma è un valore costante!

```
int verificaTavola(int N[][n], int P[]){
    int i;
    for(i = 0;  i < n; i++)
        if (C[P[i]][P[(i+1)%n]]==1)
            return 0;
    return 1;
}
```

In questa funzione viene utilizzato un nuovo costrutto, il `for`, che analogamente al `while`, permette l'esecuzione di un blocco di istruzioni fintanto che una data condizione è vera. Nel caso specifico l'istruzione inizializza la variabile `i` a 0 e fintanto che `i` è più piccolo di `n - 1` esegue l'istruzione `if` ed incrementa `i`. Formalmente l'istruzione

```
for(istruzione0;  condizione; istruzione2)
    istruzione1;
```

equivale a

```
istruzione0;
while(condizione){
    istruzione1;
    istruzione2;
}
```

e permette di rendere il codice, in alcuni casi, un po' più compatto.

Si osservi inoltre che nella descrizione degli argomenti della funzione il vettore P compare senza dimensione. Questo è corretto in quanto la variabile P contiene un indirizzo di memoria: indicare la dimensione del vettore o meno non fa alcuna differenza. Invece per quanto riguarda la matrice N si è reso necessario indicare almeno la seconda dimensione, il motivo di questo è spiegato nel paragrafo a pagina 29.

La funzione `verificaTavola` può essere utilizzata per risolvere il problema di re Artù analizzando tutte le possibili permutazioni dei cavalieri fino a trovarne una che soddisfa i requisiti (ammesso che ne esista almeno una). Tale algoritmo è schematizzabile come segue.

- Per ogni permutazione `P`
  - se `verificaTavola(N, P) == 1` termina restituendo `P` come soluzione
- Nel caso la permutazione non viene trovata concludi che i vincoli sono inconsistenti.

Nel caso in cui non esiste nessuna soluzione dobbiamo analizzare tutte le $n!$ permutazioni.

Chiediamoci ora quanto impiega il nostro programma per terminare il suo compito. Chiaramente questo è variabile e dipende da molti fattori: l'istanza di input, su quale

macchina viene fatto girare il programma, che tipo di ottimizzazioni esegue il compilatore che traduce il programma e così via. Ma il parametro più importante è n, ovvero il numero di cavalieri da far sedere. Cerchiamo di semplificare il più possibile il problema eliminando alcuni fattori che influenzano il tempo di calcolo ma che, o non sono prevedibili (come la caratteristica dell'input) o dipendenti da tecnologie (tipo di architettura, tipo di compilatore eccetera). Fondamentalmente vorremmo far dipendere questo tempo solo da n ovvero il numero di cavalieri ovvero la dimensione dell'istanza del problema.

L'indeterminatezza indotta dall'input si risolve considerando il caso peggiore ovvero quello che richiede un numero maggiore di passi. Nel nostro caso quello che ci costringe ad analizzare tutte le permutazioni. Questa scelta è giustificata dalla necessità di assicurare all'utilizzatore dell'algorimto delle performance che prescindono dal tipo di istanza ma che dipendono solo dalla grandezza di questa.

Per quanto riguarda invece l'influenza che l'architettura e altri fattori tecnologici esercita sulle prestazioni osserviamo che questi non concernono l'algoritmo in se, quindi non possono essere tenuti in considerazione quando si analizza un algoritmo che è una identità matematica. Per questo si assume che ogni operazione elementare (dichiarazione di variabile o funzione, assegnazione, test, operazioni aritmetiche, operazioni logiche, salti,...) che esegue un algoritmo richiede un tempo unitario per essere eseguita, quindi calcolare quanto tempo impiega un algoritmo per risolvere un problema si traduce nel calcolare quante operazioni unitarie vengono svolte (sempre nel caso peggiore). Formalmente possiamo far coincidere il tempo di esecuzione di un algoritmo col numero di istruzioni elementari fatte eseguire dall'algoritmo ad una Macchina di Von Neumann.

Consideriamo ad esempio la funzione `verificaTavola`: viene creata una variabile a cui viene assegnato il valore 0 (due operazioni di costo complessivo 2); per ogni ciclo del `for` si esegue una somma, un modulo ed un test (costo 3); se il test è positivo si esegue l'`if` che comporta un incremento ed un test (costo 2); se l'`if` ha esito positivo allora viene eseguito il `return` di costo 1 altrimenti si continua incrementando (somma più assegnazione) la variabile `i` (costo 2). Si osservi che il caso peggiore si ha quando la permutazione è corretta. In questo caso il vettore viene analizzato completamente ed il numero di operazioni richieste è $2+7n+1$. In definitiva possiamo affermare che il tempo impiegato dall'algoritmo `verificaTavola` è al più (o nel caso peggiore) una funzione lineare nella dimensione del vettore. Quest'ultima affermazione può essere riassunta dicendo che il tempo di esecuzione dell'algoritmo è $O(n)$ dove n è la dimensione dell'input. Torneremo su queste notazioni nelle prossime sezioni di questo capitolo.

Ora passiamo a calcolare il tempo di calcolo dell'algoritmo di Re Artù. Come visto, questo può invocare la funzione `verificaTavola` fino a $n!$ volte. Poiché $n! \leq n^n = 2^{n \log_2 n}$ e $n! \geq \left(\frac{n}{2}\right)^{n/2} = 2^{\frac{n}{2} \log_2 \frac{n}{2}}$ segue che la funzione fattoriale ha una crescita esponenziale[1].

Un algoritmo che può richiedere un numero esponenziale di passi è da considerarsi inefficiente infatti si supponga di avere un algoritmo che risolve un problema in tempo c^n dove c è una costante e n è la dimensione dell'input. Si supponga inoltre che per $n = 100$ la nostra potentissima macchina impiega un'ora, ovvero 60 minuti per risolvere il problema (è un problema molto complicato, non certo l'ordinamento), non è un tempo brevissimo ma potrebbe andarci bene. Proviamo ora ad eseguire l'algoritmo su un input grande 200, poiché $c^{200} = \left(c^{100}\right)^2$ ci aspettiamo che l'algoritmo termini dopo $60 \cdot 60 = 3600$ minuti ovvero dopo 60 ore. Lasciamo il computer a lavorare per tutto il fine settimana, il lunedi, dopo tre giorni,

[1] La stessa cosa si può verificare in modo più preciso utilizzando l'approssimazione di Stirling

$$n! \approx \sqrt{2n\pi}\left(\frac{n}{e}\right)^n$$

(si veda [3])

avremo i nostri risultati. Ma cosa succede se anziché raddoppiare la dimensione dell'input la deduplichiamo, ovvero passiamo all'ordine di grandezza successivo? Basta fare i conti $c^{1000} = \left(c^{100}\right)^{10}$ e $60^{10} = 604,661,760,000,000,000$. Questi sono minuti, che diventano all'incirca $1,150,421,917,808$ anni (mille miliardi!). Ora deve essere chiaro che non c'è salto tecnologico che possa tener testa a questa crescita. Anche se ogni anno riuscissimo a costruire computer che hanno una velocità doppia rispetto a quelli dell'anno precedente non sarebbe sufficiente. Possiamo tentare soltanto di trovare un algoritmo più veloce ma, almeno per questo problema, il compito è talmente arduo che finora non ne è stato trovato nessuno.

3.1 Una permutazione facile da trovare

Ora trattiamo un altro problema che richiede la ricerca di una permutazione di n interi ma questa volta la ricerca si può fare in modo efficiente.

Supponiamo di avere un vettore v di n elementi interi e di volerlo ordinare in modo non decrescente. Ovvero vogliamo trovare una permutazione di questo vettore, chiamiamola vo, tale che $\text{vo}[i] \leq \text{vo}[i+1]$ per ogni $i = 0, \ldots, n-2$.

Per risolvere questo problema mettiamoci nei panni di un romanziere che ama scrivere i suoi libri con la sua vecchia Olivetti Lettera 22 ha appena terminato il suo ultimo voluminoso romanzo aggiungendo l'ultima pagina dattiloscritta sopra la pila delle altre pagine. Proprio in quel momento una folata di vento sparpaglia il romanzo per tutta la stanza. Per sua fortuna aveva numerato le pagine. Cosa faremmo al suo posto per ricostituire l'ordine originale del romanzo? Di certo non adotteremmo la tecnica descritta all'inizio del capitolo. Una soluzione naturale potrebbe essere raccogliere tutte le pagine e farne una pila disordinata poi prendere una pagina alla volta dalla pila disordinata per inserirla in un'altra pila – inizialmente vuota – in modo ordinato.

Cerchiamo di descrivere meglio l'algoritmo dello scrittore disordinato. Da una parte abbiamo un pila disordinata A di n pagine numerate, dall'altra abbiamo una pila B di pagine numerate e ordinate con in cima alla pila la pagina col numero più alto. All'inizio la pila B è vuota e la pila A contiene tutto il romanzo. Prendiamo la prima pagina dalla pila A e la mettiamo sulla pila B (la pila B è composta da un unica pagina e quindi è ordinata). La seconda pagina della pila A la confrontiamo con l'unica pagina della pila B, se quest'ultima ha un numero minore mettiamo la pagina in cima alla pila B altrimenti mettiamo la nuova pagina sotto la vecchia prima pagina. In generale consideriamo la i-esima pagina dalla pila A il cui numero è p, sulla pila B abbiamo $i-1$ pagine ordinate il cui numero è $p_1, p_2, \ldots, p_{i-1}$ con $p_1 < p_2 < \ldots < p_{i-1}$. Se $p > p_{i-1}$ la nuova pagina viene messa in testa alla pila altrimenti questa viene scambiata con la pagina con numero p_{i-1} e al passo successivo si ripete la procedura con la pagina con numero p_{i-2}. Questo processo termina quando viene trovata una pagina p_j con $1 \leq j \leq i-1$ tale che $p_j < p$ oppure quando si raggiunge la fine della pila, in questo caso la nuova pagina va in fondo perché ha una numerazione minore di tutte le altre della pila B.

Utilizzando questa tecnica possiamo ordinare un vettore v di n elementi interi[2] nel seguente modo:

insertionSort(v)

1. per ogni $i = 1, \ldots, n-1$
   (a) $j \leftarrow i$

[2]Il tipo degli elementi non è rilevante importa solo che su questi sia definito un ordinamento totale

(b) fintanto che $j > 0$ e $v[j] < v[j-1]$
   i. scambia $v[j]$ con $v[j-1]$
   ii. $j \leftarrow (j-1)$

2. restituisci il vettore v ordinato

Questo algoritmo ha la proprietà che, per ogni $i = 0, \ldots, n-1$, il sotto-vettore $v[0], \ldots, v[i-1]$ è già ordinato (l'analogo della pila B) e che la restante parte del vettore è da ordinare (l'analogo della pila A). Ora dimostreremo la correttezza dell'algoritmo.

Lemma 3.1 (Invariante **insertionSort**). *Per $i = 1, \ldots, n-1$, alla fine dell'i-esima esecuzione del passo 1 dell'algoritmo* **insertionSort** *si ha che $v[0] \leq v[1] \leq \ldots \leq v[i]$.*

Dimostrazione. Dimostriamo il risultato per induzione su i. Per $i = 1$, $j = 1$, se $v[1] \geq v[0]$ non succede nulla, si raggiunge la fine del passo 1 ed effettivamente il sotto-vettore $v[0]$, $v[1]$ risulta ordinato. Altrimenti, se $v[1] < v[0]$ questi elementi vengono scambiati, j diventa 0 e si raggiunge la fine del passo 1 con il sotto-vettore ordinato.

Supponiamo di essere all'inizio della i-esima esecuzione del passo 1, per ipotesi induttiva sappiamo che il sotto-vettore $v[0], \ldots, v[i-1]$ è ordinato. All'interno del passo 1b l'elemento inizialmente in posizione j (originariamente in posizione i) viene di volta in volta scambiato con l'elemento che lo precede fino a che l'elemento raggiunge la prima posizione oppure l'elemento che lo precede è minore o uguale a $v[j]$. Da questa osservazione segue che al temine del passo 1b tutti gli elementi che seguono $v[j]$ (se ce ne sono) sono più grandi di $v[j]$ e che l'elemento che lo precede (se c'è) è più piccolo o uguale a $v[j]$. Dall'ipotesi induttiva e dal fatto che gli elementi $v[0], \ldots, v[i]$ hanno mantenuto la loro posizione relativa si ha la tesi. □

Proposizione 3.1. *L'algoritmo* **insertionSort** *è corretto, ovvero ordina il vettore v in modo non decrescente. Inoltre il numero di operazioni elementari che esegue nel caso peggiore è una funzione quadratica nel numero di elementi del vettore.*

Dimostrazione. La correttezza dell'algoritmo è diretta conseguenza del Lemma 3.1: quando $i = n - 1$ il vettore risulta ordinato.

Per calcolare il numero di operazioni elementari che esegue l'algoritmo cominciamo con l'osservare che questo è una funzione lineare del numero di scambi nel passo 1(b)i infatti per ogni scambio che viene effettuato si eseguono un numero costante di operazioni. Quindi il calcolo del numero di totale di istruzioni eseguite dall'algoritmo si riconduce a calcolare il numero di scambi eseguiti nel caso peggiore.

Fissato i, il numero di volte che viene eseguito lo scambio all'interno dell'i-esima esecuzione del passo 1b è al più i. Quindi il numero complessivo di scambi è

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

che è proprio una funzione quadratica nel numero di elementi del vettore. Concludiamo che, se $T(n)$ è il numero di istruzioni eseguite dall'algoritmo **insertionSort** per ordinare un vettore di n elementi allora

$$T(n) \leq cn^2 + d \tag{3.2}$$

dove c e d sono costanti. □

Il risultato appena dimostrato è frutto di una stima per eccesso riassunta nell'Equazione (3.2). È lecito chiedersi quanto questa stima sia precisa: ovvero esistono delle istanze del vettore di input che richiedono esattamente un numero di passi quadratico nella dimensione del vettore? La risposta è si in quanto se il vettore è ordinato al contrario (ovvero in modo decrescente) ogni nuovo elemento in posizione i deve essere portato – con i scambi – in posizione 0.

Ora che ci siamo assicurati che, non solo l'algoritmo **insertionSort** è corretto, ma che è anche decisamente più efficiente dell'algoritmo proposto all'inizio del capitolo, possiamo eleggerlo come algoritmo di riferimento per ordinare e quindi può valer la pena implementarlo in linguaggio C.

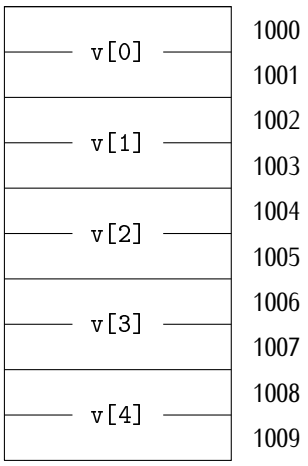
```
void insertionSort(int v[], int n){
    int i,j,t;
    for(i = 1; i < n; i++){
        j = i;
        while(j > 0 && v[j] < v[j-1]){
            t = v[j];
            v[j] = v[j-1];
            v[j-1] = t;
            j--;
        }
    }
}
```

La funzione prende in input il vettore `v` da ordinare e la sua dimensione. Il tipo di ritorno è `void`, un tipo particolare che viene usato come jolly o per indicare che la funzione non restituisce alcun risultato, come in questo caso. Se non viene indicato nessun tipo di ritorno per la funzione si assume che il tipo sia `int`.

La funzione non restituisce nulla all'esterno perché modifica il vettore `v` cambiandone l'ordine degli elementi. Si era detto nel Capitolo 2 che le modifiche apportate alle variabili locali di una funzione (tra queste ci sono anche le variabili contenenti gli argomenti della funzione) non hanno alcun impatto all'esterno della funzione in quanto queste vengono create al momento dell'ivocazione della funzione e vengono eliminate all'uscita dalla funzione. Nel nostro caso l'argomento della funzione è un vettore, ovvero l'indirizzo di memoria di un vettore creato nella funzione chiamante (presumibilmente il `main`). Questo indirizzo è il valore della variabile locale `v` ma si riferisce ad un vettore creato all'esterno. Quindi le modifiche apportate al vettore `v` nella funzione **insertionSort** in realtà sono modifiche al vettore creato esternamente.

Quanto costa accedere ad un elemento di un vettore. In questa sezione abbiamo assunto in maniera implicita che il costo per accedere ad un elemento di un vettore (ovvero l'operazione `v[i]`) è costante. Per capire come mai questa cosa è vera dobbiamo chiarire come viene memorizzato un vettore e dare qualche particolare in più sulle operazioni elementari che stanno dietro l'istruzione `v[i]`.

Al momento della definizione del vettore `v` di dimensione `N` viene allocata la quantità di memoria sufficiente a contenere tutti gli elementi del vettore, inoltre le celle di memoria allocate sono consecutive. Supponiamo, per esempio, che un `int` occupi 4 byte e che ogni parola del calcolatore sia composta da 2 byte allora la creazione di un vettore di 5 interi (`int`) produrrà la configurazione di memoria in figura.



A destra è indicato l'indice della parola di memoria. Ogni elemento del vettore occupa 4 byte quindi 2 parole di memoria. La variabile v contiene l'indirizzo (o *puntatore*) della prima parola del vettore, ovvero 1000, l'elemento `v[0]` si trova a partire dalla locazione in v, mentre l'elemento i del vettore è memorizzato a partire dalla posizione $v + 2i$. In generale, se ogni elemento del vettore v occupa c parole di memoria, l'elemento in posizione i (con $0 \leq i < N$) è memorizzato a partire dalla posizione $v + ci$.

Quindi, per accedere ad un elemento di un vettore è sufficiente eseguire una moltiplicazione ed una somma, ovvero un numero costante di operazioni elementari.

Le righe di una matrice A di dimensione $n \times m$ vengono memorizzare una dopo l'altra a partire dalla posizione di memoria contenuta in nella variabile A che rappresenta la matrice. Questo comporta che la riga i-esima della matrice è memorizzata a partire dalla posizione $A + i \cdot m \cdot d$ dove d indica quanto occupa un singolo elemento della matrice: l'elemento $A[i][j]$ è memorizzato nelle d parole di memoria a partire dalla posizione $p = A + i \cdot m \cdot d + j$. Quindi, anche in questo, caso con un numero costante di operazioni riusciamo ad accedere a qualsiasi elemento del vettore.

Si osservi ora che il valore di p dipende da m ovvero la seconda dimensione della matrice. Nel caso in cui la matrice venga usata come argomento in una funzione questo valore deve essere fornito in maniera esplicita alla funzione stessa in quanto essa non è in grado di ricavarlo. Per questo motivo nella funzione `verificaTavola` descritta all'inizio di questo capitolo abbiamo indicato la dimensione delle righe della matrice N.

3.2 Complessità computazionale

Dato un algoritmo, possiamo chiederci qual è il costo computazionale di questo, dove per costo computazionale non si intende necessariamente la quantità di tempo utilizzata per la computazione ma può trattarsi di una misura per qualsiasi risorsa utilizzata dal calcolatore: per esempio la memoria utilizzata. Col termine *complessità computazionale* di un algoritmo si intende la quantità di risorse che l'algoritmo utilizza durante la sua esecuzione. In particolare la *complessità temporale* misura il tempo di calcolo mentre la *complessità spaziale* misura la quantità di memoria. Per ora ci occuperemo di complessità temporale quindi a volte ci capiterà di omettere il termine "temporale".

Nel paragrafo precedente abbiamo visto come misurare il tempo di esecuzione – quindi la complessità temporale – di un algoritmo tenendo conto soltanto della dimensione dell'istanza. Sostanzialmente, usando l'istanza peggiore, si conta il numero di operazioni elementari

eseguite dall'algoritmo trascurando le costanti additive e moltiplicative. Ovvero si considera soltanto l'ordine di grandezza della funzione che determina il tempo di calcolo. Per esempio, se il tempo di esecuzione di un algoritmo su istanze grandi n è $4n^2 + 19n$ diciamo che l'algoritmo ha un tempo di calcolo quadratico, ovvero n^2 perché $4n^2 + 19n \leq 19n^2 + 19n \leq 38n^2$, trascurando la costante moltiplicativa resta solo n^2. In questo caso si dice che il tempo di esecuzione dell'algoritmo è $O(n^2)$ (si legge o-grande di n^2).

Definizione 3.1. *Date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$, si dice che $f \in O(g)$ se esistono due costanti c e n_0 tali che per ogni $n \geq n_0$, $f(n) \leq cg(n)$.*

Se il numero di passi eseguito dall'algoritmo A è $T(n)$ dove n è la dimensione dell'input, diciamo che A ha complessità temporale $O(f(n))$ se $T \in O(f)$. Dalla definizione appena data si deduce che la complessità computazionale è un concetto asintotico in quanto la proprietà della Definizione 3.1 vale da un certo punto in poi (la costante n_0 della definizione).

La notazione $O(.)$ ci permette di affermare che la complessità temporale di un algoritmo è al più una qualche funzione della dimensione dell'input. Tale funzione è stata trovata usando delle maggiorazioni, anche grossolane, che potrebbero portare ad un risultato notevolmente lontano da quello effettivo. Non è stato questo il caso nell'analisi dell'algoritmo **insertionSort** in quanto, non solo avevamo dimostrato una complessità $O(n^2)$, ma abbiamo dimostrato che esistono istanze che richiedono almeno un numero di passi proporzionale a n^2. In quseto caso si dice che la complessità temporale di **insertionSort** è $\Omega(n^2)$.

Definizione 3.2. *Date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$, si dice che $f \in \Omega(g)$ se esistono due costanti c e n_0 tali che per ogni $n \geq n_0$, $f(n) \geq cg(n)$.*

Inoltre se $f \in O(g)$ e $f \in \Omega(g)$ di dice che $f \in \Theta(g)$. Quindi per quanto riguarda l'algoritmo **insertionSort** possiamo affermare quanto segue.

Sia $T(n)$ il numero di passi eseguito dall'algoritmo **insertionSort** nel caso peggiore su istanze di n elementi, allora per quanto detto nel paragrafo precedente, $T(n) \in \Omega(n^2)$ (il tempo per ordinare una sequenza ordinata in modo contrario a quello voluto) quindi vale il seguente risultato.

Teorema 3.1. *La complessità temporale dell'algoritmo* **insertionSort** *è $\Theta(n^2)$.*

Si supponga che lo stesso problema P sia risolvibile da due algoritmi A_1 e A_2, il primo di complessità $\Theta(n^2)$ ed il secondo di complessità $\Theta(n^3)$, possiamo affermare che l'algoritmo A_1 è sempre migliore dell'algoritmo A_2? La risposta è no in quanto, non tenendo conto delle costanti nascoste dalla notazione Θ, potrebbe succedere che per n non molto grande A_2 sia da preferire rispetto A_1. Sappiamo invece che A_1 è più veloce di A_2 a partire da un certo n in poi, ovvero per n che tende ad infinito, ovvero A_1 è *asintoticamente* più veloce di A_2.

Supponiamo di avere due computer M_1 ed M_2, il primo 10 volte più veloce del secondo. Sul primo facciamo girare l'algoritmo A_2 (di complessità $\Theta(n^3)$ e sul secondo facciamo girare l'algoritmo A_1 (di complessità $\Theta(n^2)$). Da quanto detto precedentemente deve essere ormai chiaro che, per input abbastanza grandi, l'algoritmo più veloce sulla macchina più lenta ha prestazioni migliori dell'algoritmo più lento fatto girare sulla macchina più veloce. In definitiva questo ci dice che la misura introdotta è effettivamente una misura dell'efficienza degli algoritmi essendo indipendente da fattori esterni come il computer su cui l'algoritmo gira.

Capitolo 4

Il problema dell'ordinamento

Nel capitolo precedente abbiamo affrontato il problema dell'ordinamento progettando un algoritmo che risolve il problema in tempo $\Theta(n^2)$, dove n è il numero di elementi da ordinare. È lecito chiedersi se è possibile fare di meglio. Questo è quanto vedremo in questo capitolo.

4.1 L'algoritmo Merge-Sort

Supponiamo che il vettore di n interi v sia diviso in due parti della stessa dimensione e che queste sue parti siano ordinate. È utile questa proprietà per ordinare completamente il vettore più efficientemente rispetto all'algoritmo **insertionSort**?

Il vettore v contiene n interi tali che: per ogni $i \in \{0, \dots, n/2-2\}$, $v[i] \leq v[i+1]$ e per ogni $j \in \{n/2, \dots, n-2\}$, $v[j] \leq v[j+1]$. Sia v' il vettore contenente la permutazione di v in cui tutti gli elementi sono ordinati, allora $v'[0] = \min\{v[0], v[n/2]\}$. L'elemento successivo di v' dipende dall'elemento in posizione 0. Se in $v'[0]$ viene messo $v[0]$ allora l'elemento $v'[1] = \min\{v[1], v[n/2]\}$, altrimenti se in $v'[0]$ viene messo $v[n/2]$ allora $v'[1] = \max\{v[0], v[n/2+1]\}$. In generale assumiamo che in v' siano stati correttamente inseriti i primi i elementi di $v[0], \dots, v[n/2-1]$ ed i primi j elementi di $v[n/2], \dots, v[n-1]$ allora il prossimo elemento in v' sarà $\min\{v[i], v[j]\}$. Queste considerazioni portano all'algoritmo che segue.

merge(v)

1. $i \leftarrow 0$, $j \leftarrow n/2$, $k \leftarrow 0$
2. fintanto che $i \leq n/2-1$ e $j \leq n-2$
   (a) se $v[i] \leq v[j]$
   $v'[k] \leftarrow v[i]$, $i \leftarrow i+1$
   altrimenti
   $v'[k] \leftarrow v[j]$, $j \leftarrow j+1$
   (b) $k \leftarrow k+1$
3. se $i \leq n/2-2$, copia $v[i], \dots, v[n/2-1]$ in v' a partire da $v'[k]$
   altrimenti copia $v[j], \dots, v[n]-1$ in v' a partire dal $v'[k]$

Lemma 4.1. *Sia v un vettore di n interi tale che i due sotto-vettori $v[0], \dots, v[n/2-1]$ e $v[n/2], \dots, v[n-1]$ risultino ordinati in modo non decrescente. L'algoritmo* **merge** *con input il vettore v restituisce un ordinamento v' del vettore v. La complessità computazionale dell'algoritmo è $\Theta(n)$.*

Dimostrazione. La correttezza dell'algoritmo segue immediatamente dall'osservare che per ogni $k \in \{0, \dots, n-1\}$ viene messo in $v'[k]$ l'elemento minimo tra quelli in $v[i], \dots, v[n/2-1]$ e $v[j], \dots, v[n-1]$. Se assumiamo $v[k-1] = v[i-1]$ allora $v[k] = \min\{v[i], v[j]\}$ allora $v[i-1] \leq v[j]$ (altrimenti $v[k-1] = v[j]$) quindi $v[k-1] \leq v[i], v[j]$, ovvero $v[k-1] \leq v[k]$.

Per quanto riguarda la complessità si assuma, senza perdita di generalità, che il ciclo al passo 2 termina perchè $i = n/2-2$, ovvero perché tutti gli elementi della prima metà sono stati inseriti in v'. Poiché ad ogni passo di questo ciclo viene incrementato i oppure j, il numero di volte che viene eseguito il ciclo è $n/2+(j-n/2) = j$. Infine, nel passo 3 vengono copiati i restanti $n-j$ elementi di v in v'. Quindi il numero complessivo di operazioni è $\Theta(n)$. □

Ordinare un vettore diviso in due parti già ordinate è un problema che può essere risolto in tempo lineare nella dimensione del vettore e non quadratico. Come utilizzare questo algoritmo per ordinare un vettore qualsiasi?

4.1.1 Un algoritmo ricorsivo

Supponiamo di dover ordinare una sotto-sequenza di lunghezza ℓ, se $\ell = 1$ non c'è nulla da fare, la sequenza è già ordinata. Altrimenti dividiamo la sequenza in due parti della stessa lunghezza, ordiniamo (col metodo che stiamo descrivendo ora) le due sequenze e sulle due sequenze ordinate applichiamo l'algoritmo **merge** ottenendo l'ordinando della sequenza originale. Abbiamo appena definito un algoritmo ricorsivo, chiamato **mergeSort**. Sia v il vettore di n elementi interi e due interi l ed r tali che $0 \leq l \leq r \leq n-1$. L'algoritmo schematizzato in seguito ordina la sequenza $v[l], v[l+1], \dots, v[r]$ del vettore v.

mergeSort(v, l, r)

1. se $l \geq r$ esci
2. sia $m = (l+r)/2$
3. **mergeSort**(v, l, m)
4. **mergeSort**($v, m+1, r$)
5. **merge**(v, l, m, r)

La versione dell'algoritmo **merge** usata all'interno dell'algoritmo **mergeSort** è più generale di quella descritta. Questa ordina le due sequenze già ordinate delimitate, la prima dagli indici l e m e la seconda dagli indici $m+1$ e r (con $l \leq m \leq r$). Dopo l'esecuzione dell'algoritmo la sequenza $v[l], \dots, v[r]$ risulterà ordinata.

Teorema 4.1. *L'algoritmo* **mergeSort** *è corretto, ovvero ordina il sotto-vettore in input. Inoltre ha complessità computazionale $\Theta(n \log n)$.*

Dimostrazione. La dimostrazione di correttezza avviene per induzione sulla lunghezza della sequenza da ordinare. L'algoritmo ordina correttamente sequenze di lunghezza 0 o 1 in quanto in questo caso non fa nulla (istruzione 1). Sia $r - l + 1 = \ell$ la lunghezza di una generica sequenza di input, per ipotesi induttiva assumiamo che sequenze di lunghezza

minore di ℓ vengano ordinate correttamente. L'istruzione 3 applica l'algoritmo **mergeSort** su una sequenza lunga $m - l + 1 = (r - l)/2 + 1 < \ell$ quindi, applicando l'ipotesi induttiva, ordina la sequenza ricevuta in input. Stesso discorso vale per l'istruzione 3 che ordina la seconda parte del vettore. Infine l'applicazione **merge** successiva ordinerà le due sotto-sequenze (Lemma 4.1).

Per quanto riguarda la complessità computazionale, sia $T(n)$ il numero di operazioni elementari eseguite dall'algoritmo per ordinare un vettore di dimensione n. Siano c e d due costanti allora

$$T(n) = \begin{cases} 2T\left(\frac{n}{2}\right) + cn & \text{se } n > 1, \\ d & \text{altrimenti.} \end{cases} \tag{4.1}$$

Infatti se la sequenza è lunga 1 allora l'algoritmo termina in tempo costante. Altrimenti si esegue lo stesso algoritmo due volte su sequenze lunghe $n/2$ e successivamente si esegue l'algoritmo **merge** sulla sequenza lunga n. Il costo di questa ultima operazione è cn (Lemma 4.1). Si supponga che $n = 2^k$ (ovvero $k = \log n$) allora iterando l'Equazione 4.1 otteniamo

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{2}\right) + cn \\ &= 2\left(2T\left(\frac{n}{4}\right) + c\frac{n}{2}\right) + cn \\ &= 4T\left(\frac{n}{4}\right) + 2cn \\ &\dots \\ &= 2^i T\left(\frac{n}{2^i}\right) + icn \\ &\dots \\ &= 2^k T\left(\frac{n}{2^k}\right) + kcn \\ &= nd + cn\log n. \end{aligned}$$

Se n non è una potenza di 2 allora esiste un k tale che $n' = 2^k \leq n \leq 2^{k+1} = n''$. Allora $n'' = 2 \cdot 2^k \leq 2n$ e $n' = \frac{2^{k+1}}{2} \geq \frac{n}{2}$. Quindi $T(n) \leq T(n'') = 2nd + 2cn\log(2n) \in O(n\log n)$ e $T(n) \geq T(n') = \frac{nd}{2} + c\frac{n}{2}\log\left(\frac{n}{2}\right) \in \Omega(n\log n)$ ovvero $T(n) \in \Theta(n\log n)$. □

Per ordinare il vettore v di dimensione n si deve invocare – per esempio nella funzione `main` – la funzione **mergeSort** con parametri n, 0 e $n-1$.

4.1.2 Implementazione in C

Il linguaggio C permette la chiamata ricorsiva di funzioni, ovvero all'interno della funzione f si può invocare la funzione f su altri argomenti. Bisogna fare attenzione nell'assicurarsi che il processo di chiamate ricorsive prima o poi termini altrimenti si rischia di generare una catena infinita di chiamate alla stessa funzione. Per esempio, nell'algoritmo **mergeSort** la funzione viene chiamata ricorsivamente su sequenze sempre più corte, quindi prima o poi si arriverà ad ordinare sequenze lunghe 0 o 1 terminando il processo ricorsivo. La presenza dell'istruzione 1 gestisce questo caso e pertanto garantisce la terminazione.

La scrittura della funzione **mergeSort** è pressoché immediata, eccola.

```
void mergesort(int v[], int left, int right){
  int mid;
  if(left>=right)
    return;
  mid = (right+left)/2;
  mergesort(v, left, mid);
  mergesort(v, mid+1, right);
  merge(v, left, mid, right);
}
```

La funzione **merge** ordina i due sotto-vettori delimitati dai tre indici di input `v[left],...,v[mid]` e `v[mid+1],..., v[right]` in un vettore di appoggio `t`. Dopo questa fase viene creata l'intera sequenza ordinata `v[left],..., v[right]` copiando gli elementi di `t` in `v` a partire dalla posizione `left` di `v` (righe 20 e 21).

```
1  void merge(int v[], int left, int mid, int right){
2    int i, j, k, m = right-left+1;
3    int t[m];
4    i=left; j=mid+1;
5    k=0;
6    while(i<=mid && j<=right){
7      if(v[i] < v[j]){
8        t[k] = v[i++];
9      } else {
10       t[k] = v[j++];
11     }
12     k++;
13   }
14   while(i<=mid){
15     t[k++] = v[i++];
16   }
17   while(j<=right){
18     t[k++] = v[j++];
19   }
20   for(k=0; k<m; k++)
21     v[left+k] = t[k];
22 }
```

L'algoritmo mostrato è una generalizzazione scritta in C dell'algoritmo schematizzato all'inizio del capitolo. In questo, l'indice `i` non varia tra 0 e $n/2$ ma tra `left` e `mid` mentre l'indice j varia tra `mid+1` e `right`. Restano solo da chiarire alcuni dettagli implementativi. La prima osservazione riguarda la dichiarazione del vettore `t`. Questo viene definito di dimensione `m` dove `m` è una variabile alla quale è stato assegnato un valore nella riga precedente. Quindi è stata scelta come dimensione il valore costante che aveva la variabile `m` al momento dell'allocazione di memoria per il vettore `t`. Un eventuale cambiamento del valore di `m` non avrà alcun effetto sulla dimensione di `t` in quanto la dimensione di un vettore è costante e stabilita al momento della dichiarazione del vettore. Un'altra osservazione riguarda l'utilizzo nella stessa istruzione dell'operatore di incremento e dell'operatore di assegnazione (per esempio nella riga 8). In questo caso si legge il valore

di `v[i]`, lo si assegna a `t[k]` e alla fine si incrementa la variabile `i`. Se, anziché usare l'operatore `i++`, avessimo usato `++i` si sarebbe invertito l'ordine delle operazioni, ovvero per prima cosa si sarebbe incrementato `i` e successevamene il valore di `v[i]` sarebbe stato copiato in `t[k]`. Concludendo `t[k] = v[i++]` equivale a `t[k] = v[i]; i++` mentre `t[k] = v[++i]` equivale a `i++; t[k] = v[i]`.

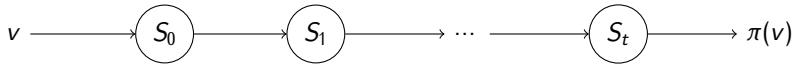
Divide & impera. L'algoritmo del **mergeSort** segue il paradigma noto sotto il nome di *divide & impera*. Questo paradigma prende il nome dalla strategia adottata già dall'Impero Romano per mantenere il predominio sui territori conquistati. Consisteva nel decentramento del potere (*divide*) al fine di governare meglio popoli conquistati (*impera*). Nella teoria degli algoritmi questa strategia si traduce nello spezzettare l'istanza del problema in istanze più piccole – e quindi più maneggevoli – e, dalle soluzioni per le istanze più piccole, ricostruire la soluzione dell'istanza di partenza. Spesso questa strategia viene implementata utilizzando la ricorsione. Nell'algoritmo del **mergeSort** dovrebbe esser chiaro qual è la parte del *divide* e quella dell'*impera*!

4.2 Ottimalità del mergeSort

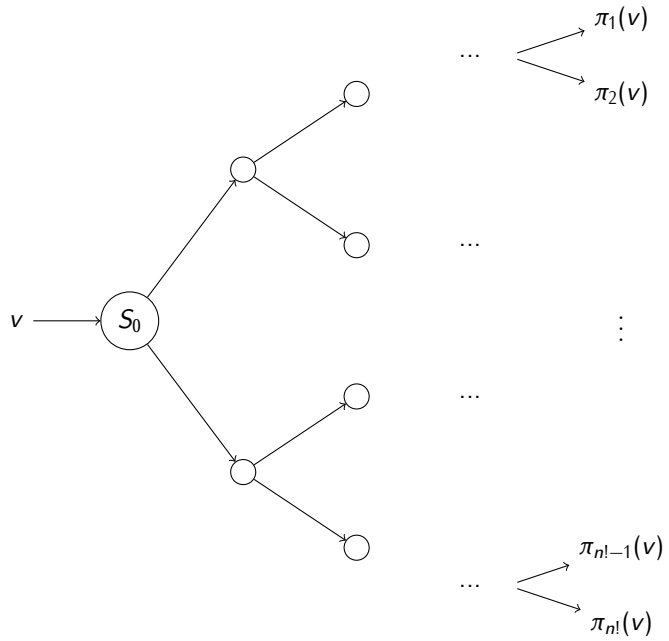
Finora abbiamo visto tre diversi algoritmi per risolvere il problema dell'ordinamento. L'algoritmo di complessità esponenziale, l'**insertionSort** di complessità quadratica ed infine il **mergeSort** di complessita $\Theta(n \log n)$. Possiamo ancora fare di meglio? Fin dove possiamo spingerci alla ricerca di algoritmi via via più efficienti? Nel caso del problema dell'ordinamento, a meno che non introduciamo vincoli sul tipo di istanza, non possiamo fare di meglio.

Teorema 4.2. *Ogni algoritmo di ordinamento (corretto) basato su confronti richiede, nel caso peggiore, tempo* $\Omega(n \log n)$ *dove* n *è la dimensione della sequenza da ordinare.*

Dimostrazione. Un qualsiasi algoritmo A di ordinamento basato su confronti prende una sequenza v di n elementi e ne restituisce una permutazione degli elementi che gode della proprietà dell'ordinamento. La computazione dell'algoritmo A può essere vista come una sequenza di stati che partono da uno stato iniziale e portano ad uno finale che individua la permutazione restituita in output. Il passaggio da uno stato a quello successivo è deciso dall'esito di un confronto.



Nell'esempio raffigurato in figura è mostrata la sequenza di stati che porta all'ordinamento della sequenza v. Lo stato iniziale S_0 è uguale per tutti mentre gli altri stati sono determinati dalle scelte dell'algoritmo in base all'input v. Poiché qualsiasi permutazione di v potrebbe essere un ordinamento, l'algoritmo deve prevedere una sequenza (a partire dalla stato iniziale S_0) verso tutti i possibili stati finali che sono $n!$ ovvero il numero di permutazioni degli elementi di v. E da ogni stato raggiunto partono 2 diramazioni, una per ogni risultato dell'operazione di confronto associata allo stato (si veda la figura seguente).



Questo diagramma rappresenta le scelte effettuate dall'algoritmo A, quindi lo possiamo chiamare D_A. Poiché siamo interessati alla complessità nel caso peggiore, dobbiamo calcolare la lunghezza ℓ_A della sequenza più lunga da S_0 ad una permutazione finale (ovvero il numero di passi eseguiti dall'algoritmo per generare questa soluzione).

Vogliamo scoprire un limite inferiore al numero di operazioni di confronto che deve eseguire, nel caso peggiore, un qualsiasi algoritmo di ordinamento. Ovvero vogliamo

$$\ell = \min_A \{\ell_A\}$$

dove il minimo viene preso su tutti gli algoritmi di ordinamento corretti.

Poiché fissato A, ℓ_A è la lunghezza del cammino massimo in D_A, possiamo assumere che tutti i cammini di D_A hanno la stessa lunghezza. Inoltre $2^{\ell_A} \geq n!$ perché ad ogni livello del diagramma al più si raddoppiano il numero di stati e deve esserci uno stato finale per ognuna delle $n!$ permutazioni. Questo deve essere vero per ogni algoritmo A. Quindi

$$2^\ell \geq n! \geq \left(\frac{n}{2}\right)^{\frac{n}{2}} = 2^{\frac{n}{2}\log\frac{n}{2}}.$$

Quindi $\ell \geq \frac{n}{2}\log\frac{n}{2}$, ovvero il numero di passi richiesto è $\Omega(n \log n)$. □

4.3 Ordinare senza confrontare

Si supponga che il vettore v che si vuole ordinare sia composto solo da interi in $\{0, 1\}$, è necessario impiegare un algoritmo di complessità $\Theta(n \log n)$? Forse possiamo sfruttare meglio la caratteristica dell'istanza per ottenere un algoritmo migliore. Iniziamo con l'osservare che una soluzione sarà composta da una sequenza di 0 seguita da una sequenza di 1. Il numero degli 0 e degli 1 sarà lo stesso del vettore di partenza v, quindi per ordinare è sufficiente contare il numero z di 0 in v e inserire nelle prime z posizioni del vettore ordinato tutti zeri, le restanti posizioni del vettore conterranno tutti 1.

binSort(v)

1. $z \leftarrow 0$
2. per $i = 0, \dots |v| - 1 = n - 1$
   se $v[i] = 0$, $z \leftarrow z + 1$
3. per $i = 0, \dots, z - 1$
   $v'[i] \leftarrow 0$
4. per $i = z, \dots, n - 1$
   $v'[i] \leftarrow 1$
5. restituisci v'

La complessità dell'algoritmo è $\Theta(n)$ in quanto nei tre cicli non si fa altro che scorrere un vettore di dimensione n.

L'algoritmo appena visto può essere generalizzato. Supponiamo che il vettore v contenga n elementi presi da un insieme di N elementi. Assumiamo pure, senza perdita di generalià, che questi siano di tipo intero nell'insieme $\{0, \dots, N-1\}$. Dobbiamo contare il numero di volte in cui compare ogni elemento nell'insieme degli elementi possibili. Per questo scopo, utilizziamo un vettore F di $N-1$ interi che contiene in posizione i la frequenza dell'elemento i in v, ovvero $F[i] = k$ se i compare k volte in v. Dopo aver costruito il vettore F, andremo a costruire il vettore ordinato inserendo prima $F[0]$ occorrenze di 0 seguite da $F[1]$ occorrenze di 1 e così via. L'algoritmo prenderà il nome di **countingSort**.

countingSort(v)

1. $F \leftarrow (0, \dots, 0)$
2. per $i = 0, \dots, n - 1$
   $F[v[i]] \leftarrow F[v[i]] + 1$
3. $k \leftarrow 0$
4. per $i = 0, \dots, N - 2$
   (a) $j \leftarrow 0$
   (b) fintanto che $j < F[i]$
   $v'[k] = i$
   $k \leftarrow k + 1$
   $j \leftarrow j + 1$
5. fintanto che $k < n$
   (a) $v'[k] = N - 1$
   (b) $k \leftarrow k + 1$
6. restituisci v'

Teorema 4.3. *L'algoritmo* **countingSort** *è corretto. Inoltre, se il vettore v in input contiene n interi appartenenti all'insieme $\{0, \dots, N-1\}$, la sua complessità temporale è $\Theta(n+N)$.*

Dimostrazione. Il passo 2 crea il vettore F delle frequenze mentre al passo 4 vengono analizzati in ordine crescente gli elementi di cui conosciamo le frequenze e vengono copiati in v'. Il passo 4b ci assicura che non tratteremo l'elemento i prima di aver inserito in v' tutti gli elementi $i-1$. Col passo 5 copiamo l'ultimo elemento $(N-1)$ nelle ultime posizioni rimaste libere di v'.

La creazione e inizializzazione del vettore F nel passo 1 richiede $\Theta(N)$ passi, mentre la costruzione del vettore delle frequenze al passo 2 ne richiede $\Theta(n)$. Per calcolare la complessità dei passi 4 e 5 relativi alla costruzione del vettore v' è sufficiente contare quante volte viene eseguito il test nella riga 4b. Fissato un elemento i questo numero è $F[i]+1$ ovvero in numero di volte in cui il risultato del test è affermativo più l'unica volta in cui è negativo. Quindi il numero di volte che viene eseguita la riga 4b è

$$F[N-1]+\sum_{i=0}^{N-2}(F[i]+1) = N+\sum_{i=0}^{N-1}F[i] = N+n.$$

Questo dimostra la tesi. □

La complessità dell'algoritmo è lineare in n e N, però non si può dire che l'algoritmo è lineare nella grandezza dell'input in quanto la grandezza dell'input dipende solo da n mentre N (anzi, $N-1$) è uno dei possibili input. Anzi, $N-1$ è proprio il massimo valore che abbiamo in input. Quando un algoritmo è lineare nel massimo valore che compare in input, esso si dice *pseudo-lineare*.

Si osservi che se $N \in O(n)$ allora il **countingSort** ha complessità lineare. In generale però non si possono fare assunzioni su quanto vale N che potrebbe anche essere esponenziale in n.

L'implementazione dell'algoritmo in C non presenta nessuna nota di rilievo né dal punto di vista progettuale né implementativo. È una semplice traduzione dall'algoritmo illustrato precedentemente.

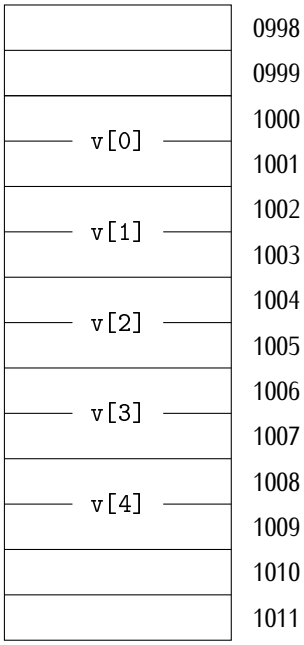
```
void countingSort(int v[], int n, int N){
     int i,j,k;
     int F[N-1];
     /* inizializzazione di F */
     for(i = 0; i < N-1; i++)
          F[i] = 0;
     /* costruzione vettore frequenza */
     for(i = 0; i < n; i++)
          F[v[i]]++;
     /* ordinamento */
     k = 0;
     for(i = 0; i < N-1; i++){
          j = 0;
          while( j < F[i] ){
               v[k++] = i;
               j++;
          }
     }
     for(;k < n; k++)
          v[k] = N;
}
```

Si osservi che all'interno del codice che descrive l'algoritmo sono presenti alcuni commenti sul codice stesso. I commenti devono essere inseriti tra i caratteri /* e */. Il compilatore ignorerà tutto quello racchiuso tra questi simboli.

Capitolo 5

Le liste

Come si è visto nella Sezione 3.1, accedere in lettura o scrittura ad un elemento del vettore ha costo costante in quanto le parole di memoria utilizzate dal vettore sono consecutive a partire da quella nota memorizzata nella variabile che identifica il vettore, ed inoltre ogni elemento del vettore occupa un numero fissato di parole di memoria. Questo sistema di memorizzazione rende complicato l'inserimento di un nuovo elemento in una data posizione del vettore. Partiamo dell'esempio della Sezione 3.1 dove un vettore di 5 interi, che occupano due parole di memoria ciascuno, viene memorizzato a partire dalla locazione 1000 della memoria.



Volendo inserire un elemento in posizione 2 bisognerebbe per prima cosa creare lo spazio tra gli attuali $v[1]$ e $v[2]$, questo comporterebbe lo spostamento di metà vettore verso la posizione 1011 oppure 0998. Questa procedura oltre a non essere efficiente a causa degli accessi in memoria che deve eseguire, è soprattutto impraticabile perché le posizioni a ridosso di quelle occupate dal vettore v (per esempio la 0999 e le 1010) potrebbero essere occupate. In questo caso l'intero vettore va memorizzata da un'altra parte in cui ci sia

lo spazio disponibile per la nuova versione del vettore v. Quello dello spazio contiguo libero potrebbe essere un grosso problema quando si ha a che fare con grosse moli di dati: potrebbe accadere che, sebbene la quantità di memoria complessiva sia sufficiente, quella contigua non sia abbastanza.

In questo capitolo introdurremo una struttura dati molto flessibile e dinamica che ci permetterà di risolvere, almeno in parte, i problemi qui sollevati.

5.1 Definizioni

Informalmente una lista è una sequenza ordinata di elementi dello stesso tipo su cui sono possibili le operazioni di lettura, inserimento, cancellazione e qualche altro. Formalmente abbiamo la seguente definizione.

Definizione 5.1. *Una lista L di elementi in X è l'insieme vuoto o un elemento di X seguito da una lista L'.*

$$L = \begin{cases} \emptyset \\ e \circ L' & \textit{dove } e \in X \textit{ ed } L' \textit{ è una lista.} \end{cases}$$

Il simbolo $\circ$ rappresenta l'operatore di concatenazione: indica che il primo operando è seguito dal secondo. Se il nostro insieme di riferimento è $\mathbb{N}$, $L = 3 \circ 5 \circ 1$ è una lista perché 1 è una lista ($e = 1$ e $L' = \emptyset$), $5 \circ 1$ è una lista ($e = 5$ e $L' = 1$) e $3 \circ 5 \circ 1$ è una lista ($e = 3$ e $L' = 3 \circ 5$).

I primi due operatori che andremo a definire permettono di isolare i due componenti di lista: l'operatore **lelem**, data una lista non vuota, restituisce il primo elemento della lista mentre l'operatore **lnext** restituisce la lista in input a meno del primo elemento.

$$\textbf{lelem}(L) = \begin{cases} e & \text{se } L = e \circ L' \\ \text{errore} & \text{altrimenti.} \end{cases}$$

$$\textbf{lnext}(L) = \begin{cases} L' & \text{se } L = e \circ L' \\ \text{errore} & \text{altrimenti.} \end{cases}$$

Un altro operatore utile permette di calcolare la lunghezza, ovvero il numero di elementi, di una lista.

$$\textbf{lsize}(L) = \begin{cases} 1 + \textbf{lsize}(L') & \text{Se } L = e \circ L' \\ 0 & \text{altrimenti.} \end{cases}$$

La dimensione di una lista è 0 se la lista è vuota, altrimenti questa sarà composta da un elemento seguito da una lista L', quindi la dimensione della lista sarà data dalla dimensione della lista L' più 1.

Data una lista $L = e_1 \circ \ldots \circ e_i \circ \ldots \circ e_{n-1}$, il prossimo operatore restituisce la sotto-lista i-esima di L ovvero $e_i \circ \ldots \circ e_{n-1}$. Sia i un intero e L una lista allora

$$\textbf{lget}(L, i) = \begin{cases} L & \text{se } i = 0 \\ \textbf{lget}(L', i-1) & \text{se } L = e \circ L' \\ \text{errore} & \text{altrimenti.} \end{cases}$$

Se i è uguale a 0 allora la lista che si ottiene è quella di partenza. Altrimenti, la lista voluta è la $i-1$-esima a partire dal secondo elemento di L. Se i è maggiore della lunghezza della lista si restituisce un errore.

Il prossimo operatore inserisce un elemento $e' \in X$ in posizione $i \in \mathbb{N}$ della lista L.

$$\textbf{linsert}(L, e', i) = \begin{cases} e \circ \textbf{linsert}(L', e', i-1) & \text{se } i > 0 \text{ e } L = e \circ L' \\ e' \circ L & \text{se } i = 0 \\ \text{errore} & \text{altrimenti} \end{cases}$$

Se $i = 0$ dobbiamo inserire in testa alla lista L, quindi la nuova lista sarà costituita dal nuovo elemento seguito dalla lista L. Se l'elemento deve essere inserito in una posizione intermedia, questa sarà la $i-1$ esima della lista L'. Come al solito se i è maggiore della lunghezza della lista si verificherà una condizione di errore. Vediamo un esempio di funzionamento di questo operatore. Sia $L = 1 \circ 2 \circ 3 \circ 4 \circ 5$, $i = 2$ e $e' = 0$ allora

$$\begin{aligned} \textbf{linsert}(1 \circ 2 \circ 3 \circ 4 \circ 5, 0, 2) &= 1 \circ \textbf{linsert}(2 \circ 3 \circ 4 \circ 5, 0, 1) \\ &= 1 \circ 2 \circ \textbf{linsert}(3 \circ 4 \circ 5, 0, 0) \\ &= 1 \circ 2 \circ 0 \circ 3 \circ 4 \circ 5. \end{aligned}$$

L'ultimo operatore consente di cancellare un elemento in posizione i della lista L.

$$\textbf{ldelete}(L, i) = \begin{cases} e \circ \textbf{ldelete}(L', i-1) & \text{se } i > 0 \text{ e } L = e \circ L' \\ L' & \text{se } i = 0 \text{ e } L = e \circ L' \\ \text{errore} & \text{altrimenti} \end{cases}$$

Anche questo operatore è definito solo su liste non vuote, quindi $L = e \circ L'$. Se $i = 0$ si elimina il primo elemento della lista e si restituisce il resto. Altrimenti, se $i > 0$ allora si cancella l'elemento in posizione $i-1$ nella lista L'. Si ottiene un errore se $i > \textbf{lsize}(L)$. Come esempio consideriamo la cancellazione dell'elemento in posizione 2 della lista $L = 1 \circ 2 \circ 3 \circ 4 \circ 5$.

$$\begin{aligned} \textbf{ldelete}(1 \circ 2 \circ 3 \circ 4 \circ 5, 2) &= 1 \circ \textbf{ldelete}(2 \circ 3 \circ 4 \circ 5, 1) \\ &= 1 \circ 2 \circ \textbf{ldelete}(3 \circ 4 \circ 5, 0) \\ &= 1 \circ 2 \circ 4 \circ 5. \end{aligned}$$

5.2 Implementazione

Vorremmo implementare le liste e gli operatori descritti nella sezione precedenti in modo efficiente. Le operatori che abbiamo definito, in particolare l'operatore **linsert** e **ldelete**, ci inducono alcune scelte. Sarebbe opportuno, infatti, evitare di pretendere che gli elementi della lista siano memorizzati, come avviene per i vettori, in parole di memoria consecutive. In questo modo, le su citate operazioni possono essere implementate senza dover ricompattare la memoria allocata alla struttura. Quindi ogni elemento della lista deve poter essere memorizzato in una qualsiasi posizione di memoria. Il problema è come garantire l'accesso agli elementi della lista. Il modo comunemente utilizzato è quello di memorizzare insieme ad ogni elemento della lista la locazione di memoria in cui si trova l'elemento che lo segue. Quindi ogni elemento della lista in realtà contiene due informazioni: il dato vero e proprio e il riferimento (indirizzo di memoria) all'elemento successivo nella lista. L'ultimo elemento della lista avrà come riferimento all'elemento successivo un valore nullo.

Caccia al tesoro. È stata organizzata a Roma una caccia al tesoro con queste regole: Il punto di ritrovo, segnato su una locandina, è a Piazza della Repubblica, qui ogni concorrente

riceve l'indicazione di dove si trova il prossimo punto da raggiungere, diciamo Piazza Vittorio Emanuele. Una volta a Piazza Vittorio Emanuele, il concorrente trova l'oggetto testimone della caccia al tesoro da collezionare e l'indicazione per il punto successivo: diciamo da qualche parte in via dei Fori Imperiali. Anche qui troverà il testimone e l'indirizzo del punto successivo. Il concorrente va avanti in questo modo, di punto in punto, raccogliendo testimoni passando per Piazza Venezia, Piazza Navona, Piazza Cavour, Piazza Del Popolo e infine Piazza di Spagna dove si prende il tempo impiegato dal concorrente e si verifica che abbia raccolto tutti i testimoni. Si veda la Figura 5.1. Supponiamo che all'ultimo momento

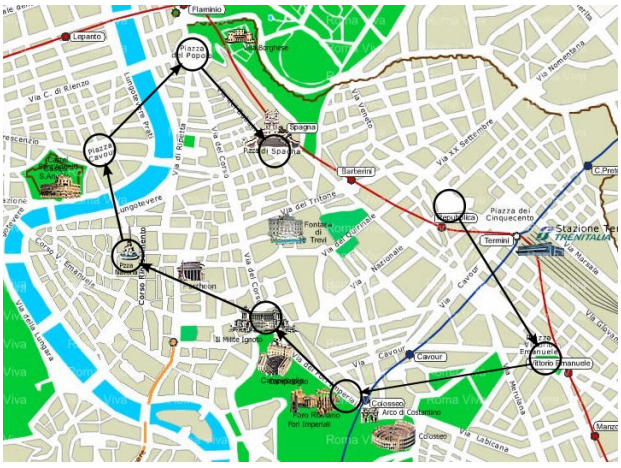


Figura 5.1: Il percorso della caccia al tesoro.

l'organizzazione decide che si vuole allungare la gara ed inserire un nuovo punto a Fontana di Trevi tra i punti Fori Imperiali e Piazza Venezia. L'organizzatore che si occupa della cosa deve: raggiungere il punto in via dei Fori Imperiali; cambiare l'indicazione del prossimo punto (non più Piazza Venezia ma Fontana di Trevi); creare il punto a Fontana di Trevi inserendo il testimone e l'indicazione per il punto successivo, ovvero Piazza Venezia. Se invece si vuole evitare la traversata del Tevere l'organizzatore deve eliminare il punto in Piazza Cavour. Quindi deve raggiungere il punto precedente a quello da eliminare e sostituire l'indicazione del punto successivo. Se l'organizzatore non sa qual è il punto successivo a Piazza Cavour, deve recarsi in Piazza Cavour, prendere l'indicazione per il punto seguente (Piazza del Popolo), tornare indietro a Piazza Navona e sostituire l'indicazione per Piazza Cavour con quella per Piazza del Popolo.

Se al posto della mappa di Roma consideriamo la memoria del calcolatore, un punto testimone/indirizzo punto successivo diventa una coppia informazione/indirizzo di memoria. Così l'indirizzo del primo punto indicato sulla locandina corrisponderà anch'esso ad un indirizzo di memoria che individuerà l'inizio della lista.

I puntatori. L'implementazione delle liste appena descritta rende necessario un meccanismo indirizzamento alla memoria "a basso livello". Il linguaggio C mette a disposizione il tipo di dato *puntatore* che serve per identificare indirizzi di memoria. Ovvero è possibile definire delle variabili il cui contenuto – il dato – è un indirizzo di memoria (o *puntatore*). Quindi un elemento della lista può essere definito come una `struct` di due campi: l'informazione ed il puntatore alla `struct` che identifica l'elemento successivo nella lista. Per dichiarare

una variabile di tipo puntatore dobbiamo specificare anche il tipo di dato a cui la variabile fa riferimento (punta). Il motivo di questo sarà chiarito in seguito. Una variabile puntatore ad un oggetto di tipo `int` si definisce con la seguente istruzione.

```
int *p;
```

Inoltre, dato un oggetto – per esempio una variabile `int` o una `struct` – possiamo risalire all'indirizzo in cui è memorizzato la attraverso l'operatore `&`. Quindi se `n` è una variabile `int`, la seguente istruzione

```
p = &n;
```

assegna a `p` l'indirizzo in cui è memorizzato `n`. Se si vuole accedere all'elemento a cui fa riferimento un puntatore si utilizza l'operatore `*`. Per esempio

```
*p = 42;
```

scrive a partire dalla locazione di memoria assegnata a `p` il valore 42. Oppure

```
printf("%d",*p);
```

stampa il contenuto dell'elemento puntato da `p`, ovvero il contenuto nell'indirizzo di memoria memorizzato in `p`. Adesso chiariremo perché al momento della dichiarazione di una variabile puntatore occorre indicare anche il tipo del dato a cui punta. Un puntatore è un indirizzo di memoria, quindi, sostanzialmente, un intero. In particolare è l'indirizzo di memoria in cui è memorizzata la prima parola del dato a cui fa riferimento. Ed è proprio in base al tipo utilizzato nelle dichiarazione del puntatore che siamo in grado sapere quante parole in tutto fanno parte del dato a cui si riferisce la variabile puntatore. Nel nostro esempio `p` è un puntatore a `int`, supponendo che un `int` occupi due parole di memoria, l'istruzione `printf` illustrata sopra prende le due parole consecutive a partire da `p`, interpreta il loro contenuto come un intero e questo verrà stampato a video. In Figura 5.2 è mostrato un esempio di una possibile configurazione della memoria ottenuta a seguito dell'esecuzione delle istruzioni che abbiamo descritto quando la variabile `n` vale 42.

Dato un puntatore di un tipo qualsiasi, diciamo il solito `int`, possiamo allocare memoria libera per questo puntatore senza dover utilizzare una variabile di appoggio come si è fatto sopra con la variabile `n`. A tale scopo si usa la funzione `malloc`. Questa funzione cerca una porzione di memoria libera la cui dimensione è determinata dal parametro in input, la alloca (quindi da questo momento in poi non è più libera) e restituisce l'indirizzo della prima parola di memoria allocata, se l'operazione non dovesse andare a buon fine restituisce un indirizzo nullo (`NULL`[1]). La dimensione della memoria desiderata deve essere indicata in byte. Per esempio, se il tipo `int` occupa 2 parole di memoria ed ogni parola è composta da 2 byte un intero viene allocato con l'istruzione che segue.

[1] Il nome `NULL` è associato ad un intero a valore negativo, solitamente -1. Si tratta di una *costante* definita per mezzo della direttiva `#define` all'interno del file `stdlib.h`. All'interno di questo file c'è un'istruzione del tipo

```
#define NULL -1
```

questo istruisce il compilatore a sostituire ogni occorrenza della parola `NULL` con il valore -1.

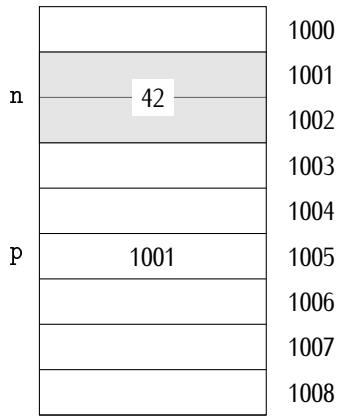


Figura 5.2: Alla variabile puntatore ad intero `p` viene assegnato l'indirizzo della variabile intera `n` che è memorizzata nelle posizioni 1001 e 1002. L'istruzione `*p = 42;` scrive 42 nella zona di memoria puntata da `p` (equivale ad assegnare 42 a `n`) e `printf("%d", *p)` stampa il valore contenuto nell'intero puntato da `p`, ovvero il valore di `n`.

```
p = malloc(4);
```

Supponendo di conoscere quanta memoria occupa un `int` e quanti byte compongono una parola di memoria possiamo utilizzare la funzione `malloc`. Tuttavia queste informazioni di cui necessitiamo sono specifiche di una architettura. Se il nostro programma deve essere compilato su due diverse architetture non possiamo utilizzare lo stesso codice. Per rendere il programma più generale si può utilizzare la funzione `sizeof`. Questa funzione restituisce il numero di byte necessari a memorizzare l'oggetto ricevuto in input. Questo può essere sia un nome di un tipo che una variabile. Quindi il modo più generale per allocare la memoria per un intero è il seguente[2]

```
p = malloc(sizeof(int));
```

Abbiamo detto che gli elementi di una lista contengono due valori: l'informazione vera e propria e l'indirizzo del prossimo elemento della lista. Poiché queste due grandezze sono disomogenee, ovvero hanno tipi diversi, il contenitore che si presta meglio per la loro memorizzazione è la `struct`. Senza perdere in generalità assumiamo che l'informazione contenuta nella lista L sia ti tipo intero, allora un elemento della lista lo possiamo definire con la `struct` che segue.

```
struct lelem {
    int info;
    struct lelem *next;
};
```

[2]Volendo essere a tutti i costi formali la sintassi più corretta è `p = (int*)malloc(sizeof(int));` In questo modo attraverso l'operatore di conversione esplicita `(·)` viene convertito il puntatore restituito dalla `malloc` in puntatore ad intero. Questo perché il tipo restituito dalla funzione `malloc` è un puntatore al tipo jolly `void`. Per approfondire l'argomento si consiglia la consultazione di [4] oppure [1].

La `struct` si compone di due campi, il campo `info` contenete l'informazione intera ed il campo `next` che è definito come un puntatore ad un'altra `struct` dello stesso tipo. Conterrà l'indirizzo dell'elemento successivo a quello in esame. Per creare un elemento della lista, quindi, possiamo usare la sintassi

```
struct lelem unelemento;
```

con le istruzioni che seguono definiamo i valori dei due campi, in particolare al campo `next` assegnamo il puntatore nullo.

```
unelemento.info = 11;
unelemento.next = NULL;
```

Facoltativamente, possiamo definire con l'istruzione `typedef` un nuovo nome di tipo da utilizzare al posto di `struct lelem`.

```
typedef struct lelem lelem;
```

Si osservi che per il nuovo tipo abbiamo usato lo stesso nome della `struct`, questa cosa è legale in quanto se ci si riferisce alla `struct lelem` allora `lelem` deve essere preceduto da `struct`. Quindi `unelemento` lo possiamo definire anche in questo modo

```
lelem unelemento;
```

Considerando la definizione ricorsiva di lista che abbiamo dato, possiamo identificare una lista con un puntatore `L` ad un `lelem` in quanto se questo puntatore è `NULL` otteniamo per `L` la lista vuota, altrimenti questo punta ad una `struct` che contiene l'informazione ed un altro puntatore a `lelem`, ovvero un'altra lista.

```
lelem *L;
L = malloc(sizeof(lelem));
(*L).info = 123;
(*L).next = NULL;
```

In questo modo abbiamo creato una lista `L` composto da un unico elemento contenente l'informazione 123. Detto ciò, siamo pronti ad implementare gli operatori definiti sella Sezione 5.1. Iniziamo dall'operatore **lelem** implementato dalla funzione `lElem`.

```
int lElem(lelem *L){
    if(L != NULL)
        return L->info;
    return ERRORE;
}
```

Se la lista `L` non è vuota viene restituito il contenuto del campo `info` della struttura puntata da `L`. Altrimenti si restituisce un errore sotto forma di un valore intero associato alla costante `ERRORE` precedentemente definita. Nella funzione si è usata la notazione `L->info` per accedere al campo della `struct` puntata dal `L`. Questa notazione è equivalente a `(*L).info` che risulta essere un pò più pesante.

Analogamente può essere definita la funzione `lNext` che restituisce la lista in input a meno del primo elemento.

Invece è istruttivo mostrare come può essere implementato l'operatore **lsize**. Di questo presenteremo due versioni: quella iterativa e quella ricorsiva. Per quanto riguarda la versione ricorsiva non dobbiamo fare altro che seguire la definizione di **lsize**.

```
int lSize(lelem *L){
    if (L == NULL)
        return 0;
    return 1 + lSize(L->next);
}
```

Nella versione iterativa dobbiamo implementare un meccanismo che scorre tutti gli elementi dalla lista con un ciclo.

```
int lSize(lele *L){
    int size = 0;
    lelem *p = L;
    while(p != NULL) {
        size++;
        p = p->next;
    }
    return size;
}
```

All'entrata della funzione viene definito l'intero `size` inizializzato a 0. Questo viene incrementato ogni volta che scopriamo l'esistenza di un nuovo elemento nella lista. Per scorrere tutti gli elementi della lista usiamo un puntatore a `lelem`, `p`. Partendo dal primo elemento della lista ci spostiamo di elemento in elemento fermandoci soltanto quando incontriamo il puntatore `NULL`. Riprendendo l'esempio della caccia al tesoro, è come se volessimo contare il numero di punti. Noi siamo `p`, partiamo dal primo punto indicato nella locandina (`p = L`), ci spostiamo verso il prossimo punto usando come indirizzo quello trovato sul punto attuale (`p = p->next`). Ogni volta aumentiamo il nostro contatore di 1 e andiamo avanti così fino ad incontrare un punto che non ne ha uno successivo (`p = NULL`).

Anche per la funzione che implementa l'operatore **lget** se ne può dare una versione ricorsiva direttamente derivata dalla definizione.

```
lelem *lGet(lelem *L, int i){
    if (L == NULL || i == 0)
        return L;
    return lGet(L->next, i-1);
}
```

Mentre la versione iterativa interrompe la scansione della lista all'i-esimo elemento restituendone l'indirizzo.

```
lelem *lGet(lelem *L, int i){
    while( L!=NULL && i > 0){
        L = L->next;
        i--;
    }
    if(i==0)
        return L;
    else
        return ERRORE;
}
```

In questa funzione è stata utilizzata la stessa variabile `L` in input per scorrere la lista. Come è stato già precedentemente fatto notare, questa cosa non ha effetti collaterali all'esterno della funzione in quanto la variabile `L` è una variabile locale della funzione contenente una copia del valore della variabile passata in input alla funzione. Lo stesso vale per la variabile intera `i`. Passando all'analisi del codice, si esce dal ciclo `while` quando si raggiunge la fine della lista oppure quando si raggiunge l'elemento in posizione i. All'uscita del ciclo, viene eseguito un controllo sulle variabili `i` e `L` per distinguere i due casi possibili: se `i` vale 0 abbiamo trovato la lista che ci interessa e la restituiamo in output, altrimenti `i` è maggiore di 0 quindi il `while` è terminato perché abbiamo raggiunto la fine della lista, ovvero non esiste nessun elemento in posizione `i` quindi viene restituito `ERRORE`.

Passiamo ora alla implementazione degli operatori più articolati **linsert** e **ldelete**.

L'operatore linsert. Innanzi tutto precisiamo che, al fine di semplificare il codice, rispetto alla definizione dell'operatore **linsert**, quando si cercherà di inserire un elemento in una posizione che va oltre la dimensione della lista, l'elemento viene aggiunto in coda.

```
lelem *lInsertFirst(lelem* L, int elem){
    lelem *t =  malloc(sizeof(lelem));
    t->info = elem;
    t->next = L;
    return t;
}

lelem *lInsert(lelem* L, int elem, int i){
    if ( i == 0 || L == NULL )
        return lInsertFirst(L, elem);
    L->next = lInsert(L->next, elem, i-1);
    return L;
}
```

La versione ricorsiva della funzione `lInsert` utilizza la funzione `lInsertFirst` che aggiunge l'elemento in testa alla coda. Questa funzione viene invocata se la coda è vuota o si vuole inserire l'elemento in posizione 0. Altrimenti (la coda è composta da almeno un elemento ed il nuovo elemento da inserire non è il primo) si inserisce l'elemento in posizione $i-1$ nella lista che inizia dall'elemento puntato da `L->next`. La lista così modificata viene indirizzata dal campo `next` del primo elemento. Torniamo alla funzione `lInsertFirst`: questa crea un nuovo elemento puntato dalla variabile `t` che dovrà diventare il nuovo primo elemento della lista, quindi il campo `next` di questo punterà ad `L` (il precedente primo

elemento della lista). Viene restituito l'indirizzo del primo elemento della lista modificata, ovvero `t`.

Veniamo alla versione iterativa.

```
1  lelem *listInsert(lelem* L, \
2                int elem, int i){
3      lelem *t, *p;
4      if(i == 0 || L == NULL){
5          t = malloc(sizeof(lelem));
6          t->info = elem;
7          t->next = L;
8          return t;
9      }
10     p = L;
11     while( p->next != NULL && i - 1 > 0 ){
12         p = p->next;
13         i--;
14     }
15     t = malloc(sizeof(lelem));
16     t->info = elem;
17     t->next = p->next;
18     p->next = t;
19     return L;
20 }
```

Ritornando alla caccia al tesoro, siamo nella condizione di dover inserire un nuovo punto in una posizione stabilita. Si noti che la procedura da seguire è diversa a seconda che il nuovo punto vada inserito in posizione 0, ovvero in testa, oppure in un altra posizione interna. Nel primo caso, una volta scelta la posizione del punto, dobbiamo cambiare l'indirizzo del ritrovo sulla locandina. Contestualmente, sul punto appena aggiunto dobbiamo indicare come punto successivo quello del vecchio ritrovo il cui indirizzo lo troviamo sulla locandina. Nel caso delle liste, inseriamo in posizione 0 se lo richiede l'utente (`i` equivale a 0) oppure se la lista è vuota. In questo caso creiamo il nuovo elemento (riga 5), nel campo `info` inseriamo l'informazione passata alla funzione (riga 6) e nel capo `next` inseriamo l'indirizzo del vecchio primo elemento della lista che troviamo nella variabile `L` (riga 7). L'indirizzo del nuovo primo elemento della lista è memorizzato nella variabile `t` che viene restituito in output (riga 8).

Se dobbiamo inserire un punto in una posizione interna del percorso della caccia al tesoro, per prima cosa raggiungiamo il punto che lo dovrebbe precedere nella nuova configurazione. Da qui infatti prendiamo l'indirizzo del punto che lo dovrebbe seguire e, sempre in questo punto, dobbiamo inserire l'indirizzo del punto nuovo arrivato. Le istruzioni nelle righe 10–14 posizionano un puntatore `p` sull'elemento della lisa che precederà quello che andremo ad inserire. Questo elemento lo troveremo scandendo la lista dal primo elemento decrementando di volta in volta il contenuto della variabile `i` fino a che `i - 1` è 0 (vogliamo posizionare `p` in posizione `i - 1`) oppure `p` raggiunge l'ultimo elemento della lista.

La Figura 5.3 rappresenta graficamente la situazione all'uscita del ciclo. Gli elementi della lista sono rappresentati con degli "scatolotti" composti da due reparti, quello relativo al campo `info` e quello al campo `next`. L'indirizzo contenuto in questo ultimo campo viene rappresentato con una freccia che arriva allo "scatolotto" a cui fa riferimento. Anche il valore della variabile `p`, essendo un puntatore, viene rappresentato con una freccia.

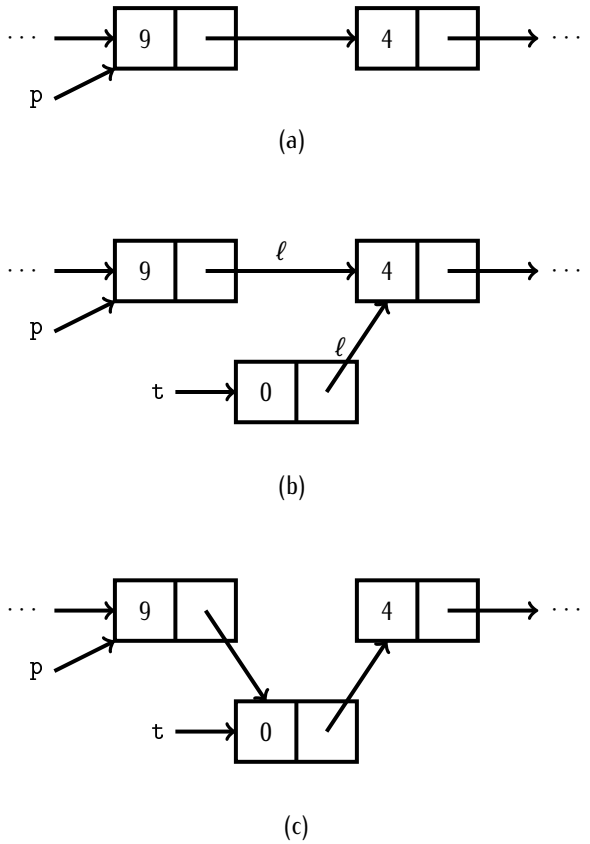


Figura 5.3: L'inserimento di un elemento in posizione `i`.

All'uscita del ciclo `while` (Figura 5.3a) viene creato il nuovo elemento il cui indirizzo viene memorizzato in `t` (Figura 5.3b). Il campo `info` del nuovo elemento sarà il valore della variabile `elem` passata alla funzione mentre il campo `next` sarà l'indirizzo di memoria che troviamo in `p->next` (l'indirizzo del vecchio elemento in posizione `i`). Per finire, nel campo `p->next` andiamo a scrivere l'indirizzo dell'elemento appena creato (Figura 5.3c). Viene restituito in output l'indirizzo del primo elemento della lista, ovvero il valore di `L`.

L'operatore ldelete. Iniziamo col dire che la cancellazione di un elemento da una lista vuota restituisce una lista vuota, mentre cancellare un elemento in una posizione inesistente non causa alcun effetto.

Vediamo prima la versione iterativa.

```
1  lelem *lDelete(lelem* L, \
2                  int i){
3      lelem *p;
4      if(L==NULL)
5          return NULL;
6      if(i==0)
7          return L->next;
8      p = L;
9      while( p->next != NULL && i - 1 > 0 ){
10         p = p->next;
11         i--;
12     }
13     if(i==1){
14         p->next = p->next->next;
15     }
16     return L;
17 }
```

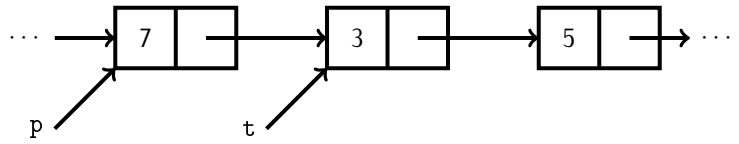
Assumiamo di avere una lista non vuota, la cancellazione del primo elemento della lista lo si ottiene restituendo in output come indirizzo del nuovo primo elemento l'indirizzo dell'attuale secondo elemento che si trova nel campo `L->next`. Nel caso in cui si debba cancellare un elemento in posizione `i` interna, come nel caso della `lInsert` si raggiunge con un puntatore `p` l'elemento in posizione `i - 1`, da qui possiamo accedere all'indirizzo dell'`i + 1`-esimo elemento (`p->next->next`) che copiamo in `p->next`. In questo modo la lista risultante non contiene più l'elemento in posizione `i`, nel senso che non è più collegato in alcun modo col resto della lista. Tuttavia la `struct` che lo definisce è ancora in memoria ad occupare spazio che non potrà essere più utilizzato fino alla conclusione del programma. Nel caso di una lista molto grande su cui si fanno molti inserimenti e cancellazioni si corre il rischio di esaurire la memoria a disposizione. Per questo motivo è buona norma liberare la memoria occupata dalla `struct` relativa ad un elemento della lista che viene cancellato. Per questo scopo si utilizza la funzione `free`. Questa funzione libera la memoria puntata dall'indirizzo che prende in input. Per introdurre questa caratteristica nella funzione, utilizziamo un nuovo puntatore `t` a `lelem` su cui salviamo l'indirizzo della `struct` che deve essere eliminata, quindi, dopo aver sistemato nel giusto ordine gli elementi della lista, la memoria puntata da `t` può esser liberata. Ecco il codice della funzione `lDelete` modificata.

```
1  lelem *lDelete(lelem* L, \
2                 int i){
3      lelem *p, *t;
4      if(L==NULL)
5          return NULL;
6      if(i==0){
7          t = L;
8          L = L->next;
9          free(t);
10         return L;
11     }
12     p = L;
13     while( p->next != NULL && i -1 > 0 ){
14         p = p->next;
15         i--;
16     }
17     if(i==1){
18         t = p->next;
19         p->next = p->next->next;
20         free(t);
21     }
22     return L;
23 }
```

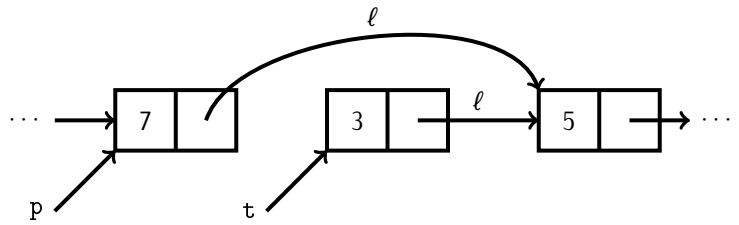
Vediamo in dettaglio quello che succede all'uscita del ciclo `while`. Dopo aver posizionato il puntatore `p` nella posizione `i - 1`, salviamo in `t` l'indirizzo dell'elemento in posizione `i` (riga 18 e Figura 5.4a); copiamo in `p->next` l'indirizzo dell'elemento in posizione `i + 1` (riga 19 e Figura 5.4b); liberiamo la memoria puntata da `p` (riga 20 e Figura 5.4c).

Per quanto riguarda la versione ricorsiva della funzione, come avveniva per l'operatore di inserimento, utilizziamo una funzione che tratta separatamente la cancellazione del primo elemento.

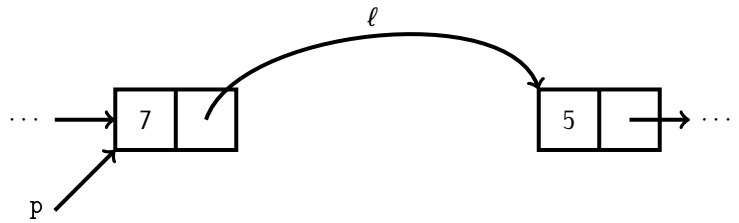
```
lelem *lDeleteFirst(lelem* L){
    lelem *t;
    t = L;
    L = L->next;
    free(t);
    return L;
}

lelem *lDelete(lelem* L, int i){
    if (L == NULL)
        return NULL;
    if (i == 0)
        return lDeleteFirst(L);
    L->next = lDelete(L->next, i-1);
    return L;
}
}
```



(a)



(b)



(c)

Figura 5.4: La cancellazione di un elemento in posizione `i`.

La funzione `lDeleteFirst` sposta il puntatore `L` sul secondo elemento della lista non prima di aver salvato l'indirizzo dell'elemento da eliminare nella variabile `t`. Il funzionamento della `lDelete` è analogo alla `lInsert`.

Completiamo la sezione mostrando il codice del programma che utilizza le funzioni appena descritte.

```
#include <stdio.h>
#include <stdlib.h>

#define ERRORE -9999

struct lelem {
    int info;
    struct lelem *next;
};
typedef struct lelem lelem;

int lSize(lelem*);
lelem *lGet(lelem*, int);
lelem *lInsert(lelem*, int, int);
lelem *lDelete(lelem*, int);
void lPrint(lelem*);

main(){
    lelem *L = NULL;
    L = lInsert(L, 0, 0);
    L = lInsert(L, 1, 0);
    L = lInsert(L, 2, 1);
    L = lInsert(L, 3, 2);
    L = lInsert(L, 4, 0);
    L = lInsert(L, 5, 10);
    lPrint(L);
    L = lDelete(L,0);
    lPrint(L);
}
```

Nella funzione `main` viene creata una lista vuota `L` alla quale vengono aggiunti e tolti degli elementi in varie posizioni. Una volta completato l'inserimento e poi la cancellazione la lista viene stampata a schermo utilizzando la funzione `lPrint` di cui non abbiamo parlato ma che, dopo tutto quanto è stato detto, dovrebbe essere facilmente implementabile dallo studente. Il codice delle funzioni non è stato riportato per motivi di spazio ma può essere inserito o prima della funzione `main` oppure dopo. Prima della definizione del `main` e dell'elenco dei prototipi delle funzioni, troviamo la definizione della `struct lelem` e del tipo omonimo. Queste due definizioni sono poste al di fuori di tutte le funzioni del programma quindi sono "visibili" a tutte le funzioni. Ovvero possono essere utilizzate all'interno delle funzioni che compaiono nel programma. Se viceversa avessimo definito la `struct lelem` all'interno della funzione `main` non l'avremmo potuta utilizzare altrove, per esempio nella funzione `lSize`. Altra cosa degna di nota è la definizione della costante `ERRORE` per mezzo della direttiva `#define`. Questa direttiva informa il compilatore che deve sostituire

tutte le occorrenze della parola `ERRORE` all'interno del programma con tutto quello che segue alla destra di `ERRORE` fino a fine riga. Per questo motivo la direttiva `#define` non viene terminata col punto e virgola, altrimenti anche quello verrebbe incluso nella definizione (anziché sostituire `ERRORE` con `-9999` verrebbe sostituito con `-9999;`). Per finire, viene incluso il file `stdlib.h` della libreria `stdlib` che contiene i prototipi delle funzioni `malloc` e `free` e la definizione della costante `NULL`. Infine viene incluso il file `stdio.h` per il prototipo della funzione `printf`.

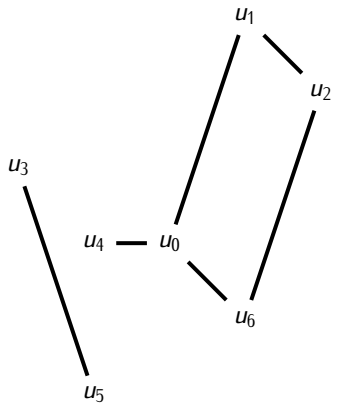
5.3 Efficienza

Come abbiamo appena visto, la struttura dati lista è molto versatile in quanto permette un utilizzo più razionale della memoria. Tuttavia, a causa della sua natura dinamica, l'accesso ad un elemento di una lista con n elementi richiede la scansione di tutti gli elementi che lo precedono. Quindi, nel caso peggiore, l'accesso ad un elemento in una determinata posizione della lista può richiedere $\Theta(n)$ passi. Abbiamo visto che la stessa operazione in un vettore richiede tempo costante. Ma nel vettore inserire e cancellare un elemento richiede la ridefinizione del vettore stesso, mentre con la lista questa è una operazione il cui costo è lineare nella dimensione della lista.

Capitolo 6

Reti e grafi

Internet, le ferrovie, il web, le autostrade, i collegamenti aerei, Facebook, gli organigrammi aziendali hanno una cosa in comune, sono reti. Una *rete* non è altro un insieme di oggetti tra i quali esiste una qualche relazione. Un esempio potrebbe essere la rete di amicizie tra n individui, una rete di questo tipo può essere rappresentata graficamente ponendo ogni individuo in una posizione del piano e, se due individui sono amici, li collegheremo con un segmento.



Dalla rappresentazione grafica deduciamo per esempio che u_0 è amico di u_1, u_4 e u_6, mentre u_3 e u_5 sono amici e non hanno nessun altro collegamento di amicizia con gli altri individui. Se u_4 da una festa ed invita u_0 chiedendogli di portare qualche amico, potrebbero arrivare anche u_1 e u_6. Inoltre se anche u_1 e u_6 fanno la stessa cosa, potrebbe essere coinvolto anche u_2. Per u_3 e u_5 non c'è speranza, trascorreranno la serata al cinema.

6.1 I Grafi

Le reti di relazioni di cui abbiamo discusso precedentemente si prestano ad essere rappresentate per mezzo dei *grafi*. Un grafo è costituito da un insieme finito di elementi chiamati *nodi* e da un insieme che rappresenta le relazioni tra coppie di nodi. Se la relazione è simmetrica (come l'amicizia), questa è rappresentata utilizzando un insieme di due elementi

distinti, altrimenti, se la relazione è asimmetrica viene rappresentata con una coppia ordinata di nodi. Nel primo caso il grafo si dice *non diretto* mentre nel secondo caso si dice *diretto*.

Definizione 6.1. *Un grafo non diretto G è definito da una coppia di insiemi (V, E) dove $V = \{v_0, \dots, v_{n-1}\}$ è un insieme finito di elementi detti nodi ed $E = \{e_0, \dots, e_{m-1}\}$ è un insieme finito di sottoinsiemi di V di cardinalià 2 detti archi. Ovvero per ogni $i \in 0, \dots, m-1$, $e_i \subseteq E$ e $|e_i| = 2$.*

Un grafo $G = (V, E)$ è diretto se $E = \{e_0, \dots, e_{m-1}\}$ e per ogni $e \in E$ vale $e = (u, v)$ con $u, v \in V$.

Il grafo che rappresenta la rete delle amicizie illustrato all'inizio di questo capitolo è un grafo non diretto in cui $V = \{u_0, u_1, u_2, u_3, u_4, u_5, u_6\}$ e

$$E = \{\{u_0, u_1\}, \{u_0, u_4\}, \{u_0, u_6\}, \{u_1, u_2\}, \{u_2, u_6\}, \{u_3, u_5\}\}\,.$$

Comunemente anche gli archi dei grafi non diretti vengono rappresentati come coppie anziché come insiemi. L'ambiguità viene eliminata specificando di volta in volta se si tratta di grafi diretti o non diretti. Anche in queste note useremo lo stesso sistema, quindi diremo che il grafo illustrato all'inizio del capitolo è un grafo non diretto che ha questo insieme di archi

$$E = \{(u_0, u_1), (u_0, u_4), (u_0, u_6), (u_1, u_2), (u_2, u_6), (u_3, u_5)\}\,.$$

Esistono molti problemi definiti su reti e quindi su grafi, basti pensare ai problemi di comunicazione sulla rete internet. In questo capitolo ne affronteremo uno in particolare: il problema della connettività. La risoluzione di questo problema astratto che definiremo tra poco ci permetterà di rispondere ad un certo numero di domande più concrete. Per esempio, nella rete dei collegamenti stabiliti da una compagnia aerea, quali città posso raggiungere partendo da Roma? Oppure, per raggiungere una determinata città qual è il percorso che mi permette di fare il minor numero di scali? Prima di affrontare questo argomento abbiamo bisogno di qualche definizione preliminare.

Dato un nodo u di un grafo non diretto $G = (V, E)$, definiamo con $N(u)$ l'insieme dei nodi di G che sono collegati a u per mezzo di qualche arco in E. Ovvero $N(u) = \{v : \exists v \in V, (u, v) \in E\}$. L'insieme $N(u)$ viene chiamato *vicinato* di u ed i suoi componenti sono i *vicini* di u. Un *cammino* tra i nodi u e v di G è una sequenza di nodi $p_{uv} = x_0, x_1, \dots, x_k$ tale che $x_0 \equiv u$, $x_k \equiv v$ e per ogni $i = 0, \dots, k-1$, $(x_i, x_{i+1}) \in E$. Quindi un cammino è una sequenza di archi adiacenti che permettono di collegare due nodi che vengono detti *connessi*. Un cammino tra due nodi è detto *semplice* se non passa due volte per lo stesso nodo. La *lunghezza* di un cammino p è data dal numero di archi che lo compongono, essa è indicata con $|p|$. Nell'esempio di inizio capitolo la sequenza u_4, u_0, u_6, u_2 è un cammino semplice tra u_4 e u_2 di lunghezza 3 quindi questi due nodi sono connessi. Un grafo si dice *connesso* se ogni coppia dei suoi nodi è connessa. Il solito grafo dell'esempio non è connesso in quanto non esiste nessun cammino tra u_3 e u_4.

6.2 Implementazioni dei grafi e matrici

Un grafo di n nodi può essere rappresentato con una matrice E di dimensione $n \times n$. Assumendo senza perdita di generalità che i nodi siano interi da 0 a $n-1$, $E[i, j] > 0$ se esiste l'arco tra il nodo i ed il nodo j, altrimenti $E[i, j] = 0$. La matrice E è nota come *matrice di adiacenza* del grafo. Si osservi che tale metodo è utile per rappresentare archi sia non diretti che diretti. Nel primo caso la matrice sarà simmetrica rispetto alla diagonale

principale. La seguente matrice rappresenta il grafo illustrato all'inizio del capitolo: la corrispondenza nodi-indici della matrice è quella più naturale.

$$\begin{pmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Di seguito viene riportato il codice C di un programma che implementa il grafo di esempio e stampa a video i suoi archi.

```
#include <stdio.h>
#define n 7

main(){
     int i,j;
     int E[n][n] = {
          {0,1,0,0,1,0,1},
          {1,0,1,0,0,0,0},
          {0,1,0,0,0,0,1},
          {0,0,0,0,0,1,0},
          {1,0,0,0,0,0,0},
          {0,0,0,1,0,0,0},
          {1,0,1,0,0,0,0}
     };
     for(i = 0; i < n; i++)
          for(j = i; j < n; j++)
               if(E[i][j] == 1)
                    printf("(%d,%d)\n", i, j);
}
```

Consideriamo la potenza ℓ-esima della matrice E, ovvero $E^1 = E$ e $E^\ell = E \times E^{\ell-1}$. Il seguente risultato ci permetterà una prima analisi sulle proprietà di connettività del grafo.

Teorema 6.1. *Sia E la matrice di adiacenza di un grafo G non diretto di n nodi e ℓ un intero maggiore di 1. $E^\ell[i,j] > 0$ se e solo se in G esiste un cammino da i a j di lunghezza ℓ.*

Dimostrazione. Dimostriamo il risultato per induzione su ℓ. Se $\ell = 1$, poiché $E^\ell \equiv E$, il risultato è vero. Consideriamo $E^{\ell+1}$ si ha

$$E^{\ell+1}[i,j] = \sum_{h=0}^{n-1} E[i,h] \cdot E^\ell[h,j] > 0$$

se e solo se esiste un $k \in \{0, \dots, n-1\}$ tale che $E[i,k] \cdot E^\ell[k,j] > 0$ quindi $E[i,k] > 0$ e $E^\ell[k,j] > 0$. Per ipotesi induttiva l'ultima disuguaglianza è vera se e solo se esiste un cammino da k a j di lunghezza $\ell-1$ che insieme all'arco (i,k) forma il cammino cercato. □

Per ogni cammino non semplice che connette i con j ne esiste uno semplice che connette gli stessi nodi. Infatti, se p_{ij} è un cammino non semplice che passa più volte per un nodo k allora $p_{ij} = i, \ldots, a, k, b, \ldots, c, k, d, \ldots, j$ dove abbiamo evidenziato la prima e l'ultima occorrenza di k. Togliendo dalla sequenza tutti gli archi dal primo k all'ultimo otteniamo la sequenza $i, \ldots, a, k, d, \ldots, j$ che è ancora un cammino da i a j in cui il nodo k compare soltanto una volta. Questo procedimento può essere ripetuto per ogni nodo che compare più di una volta nella sequenza originale ottenendo un cammino semplice da i a j.

Poiché in un grafo con n nodi la lunghezza dei cammini semplici è al più $n-1$, i nodi i e j sono connessi se e solo se esiste un ℓ tale che $E^{\ell}[i,j] > 0$ ovvero se e solo se $\sum_{\ell=1}^{n-1} E^{\ell}[i,j] > 0$ (dove la somma è quella convenzionale tra matrici). Questo ci permette di progettare un algoritmo per stabilire se un grafo è connesso o meno.

1. $M, S \leftarrow E$;
2. per $n-2$ volte
   (a) $M \leftarrow M \times E$;
   (b) $S \leftarrow S + M$;
3. se ogni elemento di S (non nella diagonale principale) è maggiore di 0, il grafo è connesso

   altrimenti è non connesso;

Ad ogni passo ℓ del ciclo la matrice M rappresenta E^{ℓ} mentre la matrice S rappresenta la sommatoria di queste ultime. La connettività del grafo viene stabilita sulla base di eventuali buchi nella matrice S: in particolare se per qualche coppia di nodi i e j, $S[i,j] = 0$ allora i e j non sono connessi quindi il grafo non è connesso.

L'implementazione in C dell'algoritmo richiede l'implementazione di due funzioni per la copia, la somma e la moltiplicazione di matrici che non fanno parte del linguaggio. Iniziamo con l'illustrare la funzione `matrixCopy` che esegue una copia di una matrice in un'altra.

```
/* B = A */
void matrixCopy(int A[n][n], int B[n][n]){
     int i,j;
     for(i = 0; i < n; i++)
          for(j = 0; j < n; j++)
                    B[i][j] = A[i][j];
}
```

Ogni elemento della matrice `A` viene copiato nella stessa posizione della matrice `B` della stessa dimensione. Si osservi che la matrice `B` deve essere stata precedentemente creata. La complessità della proceduta è $\Theta(n^2)$. La somma tra due matrici viene eseguita dalla funzione `matrixSum` che somma le matrici `A` e `B`. Il risultato dell'operazione sarà la matrice `C`.

```
/* Calcola C = A+B */
void matrixSum(int A[n][n], int B[n][n],\
                            int C[n][n]){
    int i,j,k;
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
                C[i][j] = A[i][j] + B[i][j];
}
```

Anche in questo caso la complessità dell'algoritmo è $\Theta(n^2)$. Di seguito viene riportato il codice della funzione `matrixMult` che costruisce la matrice $C = A \times B$.

```
/* Calcola C = AxB */
void matrixMult(int A[n][n], int B[n][n],\
                                 int C[n][n]){
    int i,j,k;
    for(i = 0; i < n; i++)
        for(j = 0; j < n; j++)
            for(k = 0; k < n; k++)
                C[i][j]=C[i][j]+A[i][k]*B[k][j];
}
```

Dai tre cicli `for` annidati su n elementi deduciamo che la complessità della funzione è $\Theta(n^3)$. Poiché nell'algoritmo che stabilisce se il grafo è connesso o meno l'operazione di moltiplicazione viene ripetuta $n - 1$ volte, ricaviamo che il costo di questo algoritmo in termini di tempo è $\Theta(n^4)$.

```
/* = 1 e' connesso, =0 atrimenti*/
int isConnected(int E[n][n]){
    int M[n][n], S[n][n];
    int i,j;

    matrixCopy(E, M);
    matrixCopy(E, S);
    for(i = 0; i < n-1; i++){
        matrixMult(E, M, M);
        matrixSum(M, S, S);
    }
    for(i = 0; i < n; i++)
        for(j = i; j < n; j++)
            if(S[i][j] == 0)
                return 0;
    return 1;
}
```

La complessità così alta di questo algoritmo è dovuta all'operazione di moltiplicazione di matrici. Però osserviamo che, anche se sono noti algoritmi di costo minore di $\Theta(n^3)$, non possiamo in alcun modo scendere al di sotto di n^2 in quanto qualsiasi algoritmo che esegua la moltiplicazione di due matrici deve, in ogni caso, eseguire un numero costante di operazioni

Figura 6.1:

per ogni elemento della matrice risultante: si pensi all'operazione obbligatoria di scrivere i valori negli elementi della matrice. Questo implica che l'algoritmo per a verifica della connessione deve avere un costo almeno cubico nella dimensione dei nodi. Nella prossima sezione mostreremo un algoritmo alternativo di costo quadratico.

6.3 Il problema dello shortest-path

Dato un grafo $G = (V, E)$ diretto o non diretto e due nodi s e t, il problema dello *shortest-path* o cammino minimo da s a t consiste nel cercare un cammino p che congiunge s a t che abbia lunghezza minima. La lunghezza del cammino più breve da s a t viene chiamata *distanza* da s a t (in G) e si indica con $d_G(s, t)$ o semplicemente $d)s, t)$ se il grafo in questione è chiaro dal contesto.

Al fine di risolvere questo problema si segue l'approccio di esplorare, a partire dal nodo s, tutti i nodi a distanze crescenti da s tenendo traccia dei cammini via via ottenuti. Raggiunto il nodo t l'algoritmo termina restituendo il cammino cercato.

Affinché tutto funzioni correttamente dobbiamo assicurarci di esplorare i nodi a distanza $i+1$ soltanto quando tutti i nodi a distanza i sono stati esplorati. Se questo non avvenisse, potremmo raggiungere t con un cammino lungo $i + 1$ prima di accertarci che esista un cammino di lunghezza i. Per implementare questa idea dovremmo avere un insieme di tutti i nodi a distanza i da cui estrarre il prossimo nodo u da esplorare. Se questo non è t allora dovremmo passare al prossimo. Osserviamo però che u potrebbe servire per raggiungere nodi a distanza $i+1$, quindi prima di passare al prossimo nodo da esplorare consideriamo l'insieme dei vicini di u, quelli tra questi che non sono stati ancora esplorati sono a distanza $i + 1$ e quindi devono essere esplorati dopo che lo sono stati quelli a distanza i. Supponiamo che l'insieme dei nodi a distanza i sia contenuto in una struttura lineare da cui possa accedere all'elemento u più a sinistra: se esso non è t allora, contestualmente all'eliminazione di u dalla struttura, si passano in rassegna i vicini del nodo u come potenziali nodi a distanza $i + 1$; affinché questi siano analizzati dopo i nodi a distanza i, vengono introdotti in fondo alla struttura, ovvero il più a destra possibile (si veda la Figura 6.1).

Ricapitolando, abbiamo bisogno di una struttura dati lineare (tipo lista) da cui la lettura e la cancellazione avviene da un estremo e l'inserimento da quello opposto. Una struttura di questo tipo è detta *coda*.

6.3.1 Code

Una *coda* o *queue* è una struttura di dati in cui l'ultimo elemento inserito è anche l'ultimo ad essere letto o cancellato. Come la lista e la pila possiamo definire la coda in modo ricorsivo: l'insieme vuoto è una coda; se Q' è una coda ed e è un elemento del nostro universo U, $Q = e \circ Q'$ è una coda. Inoltre si definiscono gli operatori di lettura, inserimento,

cancellazione e verifica coda vuota.

$$\textbf{qget}(Q) = \begin{cases} e & \text{se } Q = e \circ Q' \\ \text{errore} & \text{altrimenti.} \end{cases}$$

$$\textbf{qinsert}(e, Q) = \begin{cases} e' \circ \textbf{qinsert}(e, Q') & \text{se } Q = e \circ Q' \\ e & \text{altrimenti.} \end{cases}$$

$$\textbf{qdelete}(Q) = \begin{cases} Q' & \text{se } Q = e \circ Q' \\ \emptyset & \text{altrimenti.} \end{cases}$$

$$\textbf{qisempty}(Q) = \begin{cases} 1 & \text{se } Q = \emptyset \\ 0 & \text{altrimenti.} \end{cases}$$

Per l'implementazione in C potremmo utilizzare una lista ed i suoi operatori. Per quanto riguarda la funzione di cancellazione e lettura non ci sono controindicazioni ma per la funzione che implementa l'operatore di inserimento si dovrebbe utilizzare `lInsert` con argomenti la coda, l'elemento e la lunghezza della coda (per forzare l'inserimento in fondo); questa soluzione, anche se corretta, è inefficiente in quanto ogni inserimento prevede la scansione dell'intera coda pertanto la sua complessità risulta essere $\Theta(n)$ dove n è la dimensione della coda. Non è questo che vogliamo. Il nostro obbiettivo è fare in modo che anche l'operatore **qinsert** venga implementato con complessità costante al pari degli altri operatori.

Una soluzione che ci permette di ottenere questo risultato è quella di utilizzare una lista con due puntatori: quello solito che punta al primo elemento della sequenza ed uno nuovo che punta all'ultimo. Con questo accorgimento la cancellazione può essere effettuata al solito modo utilizzando il primo puntatore della sequenza, mentre per l'inserimento si utilizzerà il secondo. Cominciamo col definire un nuovo tipo per l'elemento della coda utilizzando quanto già fatto per la lista.

```
typedef listelement qelem;
```

Diversamente dalla lista, una coda non può essere identificata da un unico puntatore ma, come già notato, ne occorrono due. I due puntatori agli elementi estremi della coda faranno parte di una `struct`.

```
struct queue {
    qelem *first;
    qelem *last;
};
typedef struct queue queue;
```

In una coda vuota entrambi i puntatori saranno `NULL` mentre, in generale, il campo `first` punta al primo elemento della coda ed il campo `last` punta all'ultimo (Figura 6.2). La verifica se la coda è vuota e la lettura della coda avviene attraverso il puntatore `first`.

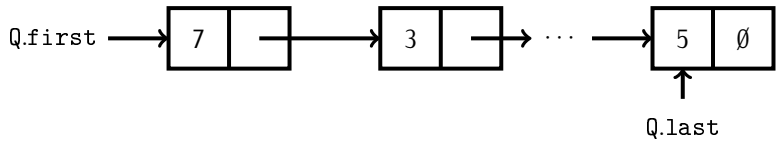


Figura 6.2: Una coda di interi.

```
int qIsEmpty(queue Q){
    if(Q.first==NULL)
        return 1;
    return 0;
}

int qGet(queue Q){
    if(!qIsEmpty(Q))
        return Q.first->info;
}
```

Anche per quanto riguarda la rimozione di un elemento dalla coda utilizzeremo l'operatore `lDelete` applicato alla testa della coda avendo l'accortezza di modificare anche il puntatore `last` nel caso in cui viene rimosso l'ultimo elemento della coda.

```
queue qDelete(queue Q){
    Q.first = listDelete(Q.first, 0);
    if (Q.first == NULL)
        Q.last = NULL;
    return Q;
}
```

Infine, l'inserimento viene eseguito utilizzando il puntatore `last`. Se la coda è vuota l'elemento inserito sarà anche il primo quindi occorre modificare anche il puntatore `first`. Altrimenti si aggiunge un elemento in posizione 1 a partire dall'ultimo (quello puntato da `last`) e si modifica `last` in maniera che punti all'elemento appena inserito.

```
queue qInsert(queue Q, int e){
    if(Q.first == NULL){
        Q.last = listInsert(Q.last, e, 0);
        Q.first = Q.last;
    } else {
        Q.last = listInsert(Q.last, e, 1);
        Q.last = Q.last->next;
    }
    return Q;
}
```

Si noti che tutti gli operatori definiti hanno complessità costante.

6.3.2 L'algoritmo per lo shortest-path

Prima di passare ai dettagli implementativi dell'algoritmo, cerchiamo prima di descriverne il funzionamento.

L'algoritmo ha come input la matrice di adiacenza del grafo E, il nodo di origine s e di arrivo t. L'output è il cammino più breve da s a t rappresentato attraverso il *vettore dei predecessori* P: in particolare sia x un nodo del grafo allora $P[x] = y$ se y precede x nel cammino più corto da s a y. Di predecessore in predecessore fino ad s siamo in grado di ricostruire tutto il cammino da t (vedere l'Esercizio 6.1). L'algoritmo mantiene un insieme X dei nodi già esplorati e ad ogni passo esso prende in considerazione un nodo u tratto da una coda Q. I nodi in Q sono connessi ad s da un cammino, al fine di poter ricostruire questo cammino abbiamo bisogno di memorizzare, insieme al nodo u anche il suo predecessore in questo cammino: sostanzialmente l'informazione contenuta negli elementi della coda sarà un arco orientato (p, u) dove p è il predecessore di u nel cammino.

- Se u è stato già esplorato (ovvero è in X) si passa a quello successivo altrimenti lo si mette in X (viene "marcato" esplorato). In ogni caso u viene eliminato dalla coda.
- Se $u \notin X$, dopo aver impostato $P[u] = p$, se u coincide con t l'algoritmo termina restituendo il cammino il vettore dei predecessori P;

  altrimenti si inseriscono in coda tutti i vicini di u e l'algoritmo prosegue col prossimo nodo in coda.

Nel caso in cui il nodo t non viene raggiunto l'algoritmo termina restituendo il cammino vuoto.

Vediamo ora come rappresentare l'input e le strutture di appoggio. Come al solito i nodi del grafo sono interi da 0 a $n-1$; l'insieme degli archi è rappresentato con una matrice di adiacenza E di dimensione $n \times n$; P è un vettore di n elementi interi, compreso -1 utilizzato per denotare un predecessore indefinito (in particolare $P[s] = -1$); l'insieme X attraverso un vettore di n elementi a valori 0 o 1, $X[u] = 1$ se e solo se u è in X; infine Q è una coda di di archi o `edge` così definiti

```
struct edge{
    int p,u;
};
typedef struct edge edge;
```

dove p è il predecessore del nodo u. La coda è composta da elementi *qelem* così definiti

```
struct qelem {
    edge e;
    struct qelem *next;
};
typedef struct qelem qelem;
```

Questa definizione comporta che le funzioni di gestione della coda devono essere opportunamente riviste. A titolo di esempio mostriamo la funzione **qget**.

```
edge qGet(queue Q){
    if(!qIsEmpty(Q))
        return Q.first->e;
}
```

Tra gli input dell'algoritmo c'è il vettore P che memorizza la soluzione trovata. L'algoritmo restituisce un intero che vale 0 se s non è connesso con t ed 1 altrimenti. In quest'ultimo caso dal vettore P si può ricostruire il cammino più breve tra i due nodi.

```
1  int shortestpath(int E[][n], int P[], int s, int t){
2      queue Q = {NULL, NULL};
3      edge e = {-1, s};
4      int X[n] = {0};
5      int v;
6      for(v = 0; v < n; v++)
7          P[v] = -1;
8      Q = qInsert(Q, e);
9      while( !qIsEmpty(Q) ){
10         e = qGet(Q);
11         Q = qDelete(Q);
12         if (X[e.u] == 0) {
13             X[e.u] = 1;
14             P[e.u] = e.p;
15             if ( e.u == t )
16                 return 1;
17             for(v = 0; v < n; v++)
18                 if (E[e.u][v] == 1){
19                     edge f = {e.u, v};
20                     Q = qInsert(Q, f);
21                 }
22         }
23     }
24     return 0;
25 }
```

Siamo pronti a dimostrare la correttezza dell'algoritmo, ovvero dimostreremo che questo restituisce il cammino di lunghezza minima tra s e t.

Teorema 6.2. *Se l'algoritmo* `shortestpath` *restituisce* 1 *il vettore P contiene il cammino minimo da s a t, altrimenti tale cammino non esiste. Inoltre l'algoritmo ha complessità $O(n^2)$.*

Dimostrazione. Iniziamo con l'osservare che se al termine dell'algoritmo per un nodo v, $P[v] = u \neq -1$ allora P codifica un cammino da s a v (a ritroso). Dimostriamo questo fatto per induzione sul numero di archi k estratti da Q. Se $k = 0$ allora per ogni nodo v $P[v] = -1$ che è coerente col nostro enunciato. Sia (u, v) il k-esimo arco estratto, questo è stato inserito in coda quando è stato estratto l'arco (p, u) per qualche nodo p. Poiché quest'ultimo arco è stato estratto prima prima di (u, v) allora vale l'ipotesi induttiva ovvero in P esiste un cammino da s ad u che con l'arco (u, v) compone un cammino da s a v.

Viceversa se s e t sono connessi allora $P[t] \neq -1$ (e l'algoritmo termina restituendo 1). Supponiamo per assurdo che $P[t] = -1$ (e quindi che l'algoritmo restituisce 0) e sia p il cammino da s a t; indichiamo con u l'ultimo nodo nel cammino a partire da s per il quale $P[u] \neq -1$ e con v il nodo che segue u (quindi $P[v] = -1$ e $X[v] = 0$). Il valore di $P[u]$ è stato definito nella riga 14 e poiché stiamo assumendo che $P[t] = -1$, $u \neq t$. Il nodo v è vicino di u quindi viene preso in considerazione nelle righe 17–21 ed inserito in coda come arco (u, v). L'algoritmo restituisce 0 quindi termina quando la coda è vuota questo implica che prima o poi l'arco (u, v) viene estratto dalla coda e visto che $X[v] = 0$

viene eseguito il corpo dell'`if` della riga 12 che tra le altre cose assegna $P[v] = u$. Ecco la contraddizione.

Ora dimostriamo che i nodi più vicini ad s vengono visitati prima di quelli più lontani, ovvero se $d_G(s,u) < d_G(s,v) = \ell$ allora il nodo u viene visitato prima del nodo v. Dimostriamo questo risultato per induzione su ℓ. Se $\ell = 1$ l'unico nodo a distanza inferiore è s che viene visitato prima di tutti i nodi. Supponiamo ora che valga l'ipotesi induttiva e siano v' e u' i nodi nel cammino più breve da s a v e da s a u che precedono, rispettivamente, i nodi v ed u. Allora $d_G(s,u') < d_G(s,v') = \ell - 1$. Poiché vale l'ipotesi induttiva u' viene visitato prima di v'. Allora u viene inserito in coda prima di v quindi u viene visitato prima di v.

Sia $p(v)$ il cammino da s a v rappresentato in P. Terminiamo la dimostrazione provando per induzione su ℓ che per ogni v tale che $d(s,v) = \ell$ e $p(v) \neq \emptyset$ allora $|p(v)| = \ell$. Se $\ell = 0$ l'unico nodo a distanza 0 da s è proprio s e $|p(u)| = 0$. Supponiamo che il risultato sia vero per tutte le distanze $d(s,\cdot) < \ell$ e che per assurdo esista un nodo v a distanza ℓ da s in G ma tale che $d(s,v) < |p(v)|$. Sia u il nodo che precede v nel cammino minimo da s a v in G e $P[v] = z$. Per ipotesi induttiva $d(s,v) = |p(u)| = \ell - 1$ inoltre $d(s,z) \geq \ell$ in quanto altrimenti, sempre grazie all'ipotesi induttiva, $|p(z)| = \ell - 1$ e di conseguenza $|p(v)| = \ell$. Per quanto affermato precedentemente, u viene visitato prima di z allora l'arco (u,v) viene introdotto in P prima che possa esser considerato l'arco (z,v) quindi $|p(v)| = \ell$.

Concludiamo la dimostrazione di correttezza osservando che se l'algoritmo restituisce 1 allora $P[t] \neq -1$ ovvero P contiene un cammino da s a t che è quello minimo. Se viene restituito 0 allora $P[t] = -1$ allora s e t non sono connessi.

Infine veniamo al calcolo della complessità dell'algoritmo. Il corpo del ciclo while viene eseguito al più m volte, ovvero tante volte quanti sono gli archi. Sia $C(e)$ il costo del blocco del while indotto dall'estrazione da Q dell'arco $e = (p,u)$.

$$c(e) = \begin{cases} O(n) & \text{se } X[u] = 0 \\ O(1) & \text{altrimenti.} \end{cases}$$

Sia E_u l'insieme degli archi che hanno come secondo estremo il nodo u, ovvero $E_u = \{e \in E : \exists x \in V, e = (x,u)\}$ allora possiamo esprimere il costo dell'algoritmo nel seguente modo.

$$\sum_{e \in E} c(e) \leq \sum_{u \in V} \sum_{e \in E_u} c(e) \leq \sum_{u \in V} (O(n) + O(1)|E_u|) = O(n^2) + O(m) = O(n^2).$$

Questo conclude la prova. □

Esercizio 6.1. *Si scriva una funzione C che prende in input il vettore dei predecessori P ed un nodo t e restituisce in output il cammino da s a t come una lista in cui il primo elemento è s e l'ultimo t.*

Esercizio 6.2. *Si progetti un algoritmo per la verifica della connettività di un grafo non diretto. L'algoritmo deve avere complessità $O(n^2)$.*

Esercizio 6.3. *La pila o stack è una struttura dati lineare sulla quale sono definite le stesse operazioni che sono definite per la coda. La differenza risiede nel fatto che l'ultimo elemento ad essere inserito è anche il prossimo elemento ad essere letto o cancellato.*

Si dia una definizione ricorsiva della struttura e delle operazioni definite su di essa. Infine si scriva il codice di tali operazioni.

Capitolo 7

Code con priorità

In una coda vige la regola che il primo che arriva è anche il primo ad essere servito. Abbiamo visto nel capitolo precedente una struttura dati che implementa questa semplice regola. Questo concetto di coda va bene per esempio all'ufficio postale o in banca ma in altre circostanze risulta essere inefficace. Per esempio nei prontosoccorso la coda viene gestita utilizzando un sistema di priorità infatti viene data precedenza ai casi più gravi.

Nella *coda con priorità* si associa ad ogni dato un peso numerico ed il prossimo elemento ad essere utilizzato è quello col peso più basso. Quindi ogni elemento nella coda porta con se due informazioni: il dato e la priorità. Senza perdita di generalità assumeremo che i due dati coincidano e che siano di tipo intero.

Le operazioni basilari che vorremmo garantire su questa struttura dati sono la lettura che ottiene l'elemento della coda con peso minimo; l'estrazione dell'elemento di peso minimo dalla coda; l'inserimento di un nuovo elemento della coda; il test che verifica se la coda è vuota o meno. Ovviamente può essere utilizzata una lista come s'è visto nel Capitolo 5 però in questo caso sia l'estrazione del minimo che la lettura del minimo costerebbe $\Theta(n^2)$ (dove n è il numero di elementi della lista) in quanto l'elemento dovrebbe essere ricercato attraverso una scansione lineare. Si noti che anche tenendo un puntatore all'elemento minimo le cose non cambierebbero in quanto anche se la lettura avverrebbe in tempo costante, l'estrazione comporterebbe la ricerca di un nuovo minimo tra gli $n-1$ elementi rimasti. Occorre inventarsi qualche cosa di diverso. Anticipiamo che la soluzione che proporremo in questo capitolo ci permetterà di limitare i costi di tutte le operazioni a $O(\log n)$.

7.1 Heap

Attraverso la struttura dati che andremo a descrivere, chiamata *heap*, quanto anticipato precedentemente sarà possibile. In particolare sarà possibile implementare l'operazione di estrazione del minimo ed inserimento di un nuovo elemento in tempo logaritmico nel numero di elementi nella struttura, mentre la lettura del minimo avverrà in tempo costante.

Gli elementi della coda vengono memorizzati in un albero: un elemento per ogni nodo dell'albero. Sia u un nodo, indicheremo con $w(u)$ il peso dell'elemento del nodo u. L'albero ha le seguenti caratteristiche.

1. L'albero è orientato verso una radice r.

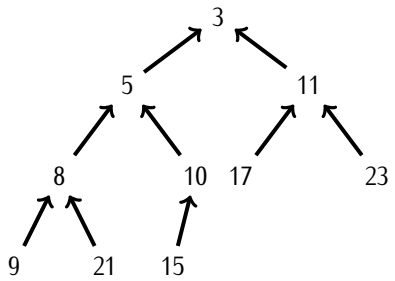
2. Ogni nodo dell'albero ha al più due figli: un albero di questo tipo si dice *binario*. I figli di ogni nodo u sono ordinati ovvero distinguiamo un figlio destro di u, $dx(u)$ ed un figlio sinistro di u, $sx(u)$.

3. Se ℓ è il nodo alla massima distanza da r (ovvero ℓ è l'*altezza* dell'albero), allora i nodi a distanza minore di ℓ costituiscono un sotto-albero binario *completo*. In un albero binario completo di altezza h ogni nodo interno (a distanza al più $h-1$ dalla radice) ha entrambi i figli mentre i nodi a distanza h non hanno figli (questi sono detti *foglie*).

4. I nodi a distanza ℓ sono il più a sinistra possibile. Un albero che rispetta 1–3 è detto *quasi completo*.

5. Per ogni nodo interno u, il peso di u è minore o uguale al peso di tutti i nodi che compongono il sotto-albero con radice u. Ovvero $w(u) \leq w(v)$ per ogni nodo v nel sotto-albero dell'albero di partenza di cui u è la radice.

Nella figura che segue mostriamo un esempio di heap. Il nodo 3 in radice è il minimo nodo dell'albero. Ogni altro nodo ha peso maggiore di tutti i nodi del sotto-albero di cui esso è radice. Inoltre l'albero è completo fino ai nodi a distanza 2 dalla radice; gli altri nodi, quelli a distanza 3, sono tutti addossati a sinistra.



Una proprietà molto importante dell'heap riguarda l'altezza dell'albero che risulta logaritmica nel numero di nodi che lo compongono. Come si vedrà dalla dimostrazione del prossimo risultato questà è una conseguenza del fatto che in un albero binario completo ogni livello[1] ha numero di nodi doppio de livello precedente.

Proposizione 7.1. *Un heap di n nodi ha altezza $\Theta(\log n)$.*

Dimostrazione. Un albero completo di altezza h ha $2^{h+1}-1$ nodi. Questo risultato può essere dimostrato per induzione su h. Se $h=0$ l'albero ha un nodo ovvero $2^{h+1}-1=1$. Il numero di foglie in un albero completo di altezza $h+1$ è 2^{h+1} in quanto il numero di nodi ad ogni livello è il doppio di quello al livello precedente. Quindi il numero di nodi in un albero completo di altezza $h+1$ è

$$2^{h+1}+2^{h+1}-1=2^{h+2}-1.$$

Se h è l'altezza di un heap di n nodi si ha

$$2^h \leq n \leq 2^{h+1}-1$$

[1] In un albero con radice il *livello k* è l'insieme dei nodi che si trovano a distanza k dalla radice.

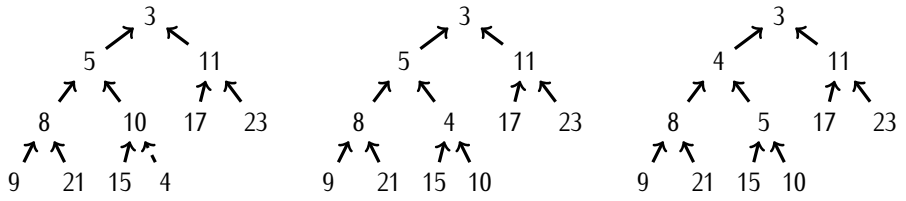
in quanto l'heap contiene almeno un nodo in più di un albero binario completo di altezza $h-1$ ed è al più un albero binario completo di altezza h. Quindi

$$h \leq \log n < h+1$$

ovvero la tesi. □

In una struttura di questo tipo è facile ottenere l'elemento minimo, in quanto si trova nella radice, e quindi l'operazione di lettura non comporta particolari problemi. Invece sia l'operazione di estrazione del minimo che quella di inserimento di un nuovo nodo devono essere tali da mantenere la struttura dell'albero coerente con le proprietà che definiscono l'heap.

Iniziamo dalla funzione **heapinsert**. Sia H un heap ed e un nuovo elemento da inserire in H il cui peso è proprio e. Prima di descrivere formalmente la funzione supponiamo che $e = 4$ deve essere inserito nell'heap nella figura sopra. Per prima cosa inseriamo il nuovo elemento sull'ultimo livello dell'albero il più a sinistra possibile (figura in basso a sinistra) in questo modo l'albero così modificato resta quasi completo ma non rispetta la 5 in quanto il nodo 10 ha peso maggiore di uno dei figli. Si osservi però che invertendo i pesi 4 e 10 il problema si risolve almeno per il sotto-albero in questione (figura in basso centrale). Però il problema si sposta sul sotto-albero la cui radice è 5. Ma l'operazione eseguita prima può essere ripetuta scambiano i pesi 4 con 5 (figura in basso a destra). Ora l'albero rispetta alche la regola 5 quindi è un heap.



Dato un nodo u dell'heap indichiamo con $f(u)$ il padre di u. Se u è la radice allora $f(u) = \epsilon$. L'inserimento di un nuovo nodo di peso e nell'heap è gestito nel seguente modo.

heapinsert(H, e)

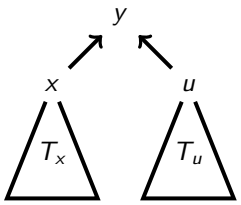
1. Aggiungi un nodo u sull'ultimo livello di H il più a sinistra possibile e sia $w(u) = e$;
2. Fintanto che $f(u) \neq \epsilon$ e $w(u) = e < w(f(u))$
   (a) scambia il peso di u col peso di $f(u)$;
   (b) $u \leftarrow f(u)$;

Ovvero, fintanto che il peso del nuovo nodo è inferiore a quello del padre, il nuovo nodo viene fatto salire di un livello. Il processo può proseguire al massimo fino a quando il nuovo nodo raggiunge la radice. Assumendo che l'operazione di inserimento di un nodo all'ultimo livello dell'heap e l'operazione $f(u)$ richieda un numero costante di passi allora vale il risultato che segue.

Teorema 7.1. *Sia H un heap di n nodi ed e un intero allora* **heapinsert**(H, e) *produce un nuovo heap in tempo, nel caso peggiore, $\Theta(\log n)$.*

Dimostrazione. L'albero resta quasi completo in quanto il nodo viene aggiunto mantenendo questa struttura mentre le successive operazioni non la modificano in quando sono semplici scambi di pesi.

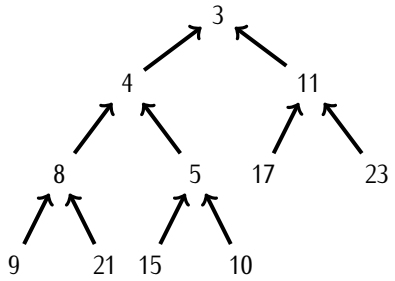
Per dimostrare che vale anche la regola 5 facciamo vedere che ogni volta che il nodo u viene fatto salire di un livello il sotto-albero di cui esso è radice è un heap. Dimostreremo questo risultato sul numero di volte h che u viene fatto salire di livello ma prima qualche notazione: sia $y = f(u)$ (ovvero il nodo y è il padre di u), x l'altro figlio di y, T_u il sotto-albero con radice u e T_x il sotto-albero con radice x. Per $h = 0$ u è appena stato inserito in H quindi la proprietà è vera.



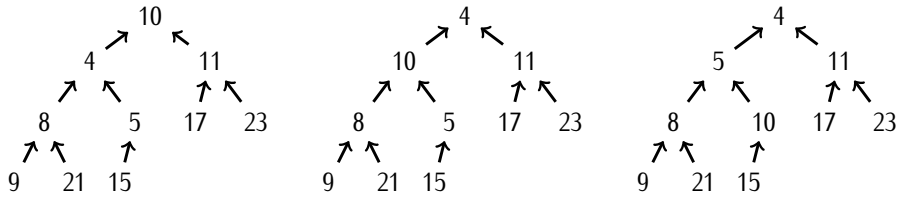
Dopo h passi sia T_u l'albero con radice u, per questo vale la regola 5. Inoltre per ogni $z \in T_u \cup T_x \cup \{x\}$ si ha $w(y) \leq w(z)$ (perché H era un heap). Se il nodo u sale di un livello allora u e y vengono scambiati perché $w(u) < w(y) \leq w(x)$ quindi la regola 5 vale per u ma anche per y.

Se si assume che tutte le operazioni eseguite dall'algoritmo richiedono un tempo costante, la complessità dell'algoritmo risulta essere proporzionale al numero di volte che il nodo u viene fatto salire di un livello. Questo valore è a sua volta proporzionale all'altezza dell'albero ovvero, dalla Proposizione 7.1, a $\log n$. □

La funzione **heapdel**(H) cancella il minimo dall'heap H e manipola la struttura rimanente in modo che questa sia torni ad essere un heap. Come al solito partiamo da un esempio applicando la procedura all'heap nella figura che segue.



Eliminiamo il minimo sostituendolo col nodo sull'ultimo livello più a destra (figura in basso a sinistra). L'albero ottenuto è ancora quasi completo ma non è un heap a causa dell posizione del nodo 10. Però osserviamo che se scambiamo il peso del nodo 10 col minimo tra i sui figli, il problema si sposta di un livello più in basso (figura in basso al centro).



Ripetendo questa procedura ancora una volta otteniamo un heap (figura in alto al destra). Ecco la descrizione formale dell'algoritmo.

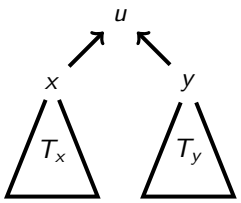
heapdel(H)

1. Sia r la radice dell'heap e v il nodo sull'ultimo livello di H il più a destra possibile: $w(r) = w(v)$;
2. elimina v;
3. $u \leftarrow r$;
4. Fintanto che $w(u) > w(sx(u))$ oppure $w(u) > w(dx(u))$
   (a) sia v il nodo di peso minimo tra $sx(u)$ e $dx(u)$;
   (b) scambia il peso di u col peso di v;
   (c) $u \leftarrow v$;

Teorema 7.2. *Sia H un heap di n nodi allora* **heapdel**(H) *produce un nuovo heap in tempo, nel caso peggiore, $\Theta(\log n)$.*

Dimostrazione. Come per il Teorema 7.1 occorre dimostrare che al termine della procedura vale la regola 5 in quanto eliminando il nodo più a destra sull'ultimo livello l'albero resta quasi completo.

Per dimostrare questo sia u la radice di un sotto-albero T_u di H e x ed y i figli sinistro e destro di u. Inoltre chiamiamo T_x e T_y gli alberi con radice x e y.



Supponiamo che T_x e T_y sono heap e che $w(u) > w(x)$ oppure $w(u) > w(y)$. Senza perdita di generalità assumiamo che $w(x) < w(y)$ allora scambiano il peso di u con quello di x si ha che T_u è un heap se e solo se T_x è un heap. Infatti se T_u è un heap allora, per definizione, lo è anche T_x. Mentre se T_x è un heap lo è T_u perché $w(u) \leq w(x), w(y)$. Il processo di portare il peso di u verso il basso prima o poi termina con un albero T_x che è un heap (nel caso estremo potrebbe essere composto da un unico nodo).

Come prima, la complessià dell'algoritmo è proporzionale all'altezza dell'albero. □

7.2 Implementazione

L'heap è un albero e come tale può essere implementato. Tuttavia, vista la sua natura si presta ad essere rappresentato mediante un vettore. Infatti ogni livello interno k è completo e quindi composto da 2^k nodi. Se i nodi venissero memorizzati nel vettore per livelli – ovvero prima la radice, poi i due nodi al livello 1 e così via – sapremmo quali sono gli indici del vettore estremi del livello. In particolare se la radice è memorizzata in posizione 1 allora il k-esimo livello inizia dall'indice 2^k e termina con l'indice $2^{k+1} - 1$. Inoltre possiamo fare in modo di ordinare i nodi del livello $k + 1$ in modo che sia possibile risalire al padre nel livello k. In particolare se un nodo si trova in posizione j allora i suoi figli (se esistono) verranno memorizzati in posizione $2j$ e $2j + 1$ (il sinistro in posizione pari

mentre il destro in posizione dispari). Si osservi che in questo modo se j varia tra 2^k e $2^{k+1} - 1$ (ovvero al livello k) allora i figli sinistri potranno variare tra 2^{k+1} e $2^{k+2} - 2$ mentre i destri tra $2^{k+1} + 1$ e $2^{k+2} - 1$. Utilizzando questa rappresentazione, in tempo costante, si può accedere al minimo dell'heap (in quanto si troverà in posizione 1), si può accedere ai figli o al padre di un nodo (e quindi scambiarne i valori) e si può inserire un nodo sull'ultimo livello il più a sinistra possibile (basta aggiungere l'elemento nella prima posizione non utilizzata del vettore).

L'heap è una struttura di dimensione variabile quindi, a priori, non sappiamo quanti elementi la compongono. Tuttavia, se la vogliamo implementare con un vettore, abbiamo bisogno di specificarne la dimensione massima raggiungibile in modo da garantire una capienza sufficiente. Quindi utilizzeremo un vettore di una dimensione che è una sovrastima dell'effettivo utilizzo di questo. Visto che il vettore non sarà mai utilizzato nella sua interezza abbiamo bisogno di conoscere il numero di elementi effettivamente utilizzati. Un heap è quindi definito da due elementi: un vettore di dimensione c (dove c è la capacità del vettore) ed un intero n che specifica il numero di elementi nell'heap. Sceglieremo c in modo da garantire $c > n$. Essendo definito come una coppia di elementi disomogenei, definiremo l'heap utilizzando una `struct`.

```
struct heap {
    int v[c];
    int n;
};
typedef struct heap heap;
```

Dove c è definita come una costante. Il vettore v è utilizzato a partire dall'indice 1 in quanto se la radice fosse memorizzata in posizione 0 il suo figlio sinistro avrebbe anch'esso indice 0. Il valore n indica il numero di elementi dell'heap ma anche la posizione dell'ultimo elemento ovvero della foglia più a destra. Per comodità definiamo una funzione che crea un heap di capacità c inizializzandone anche il campo n.

```
heap heapNew(){
    heap H;
    H.n = 0;
    return H;
}
```

La funzione che restituisce il minimo dell'heap non deve far altro che ritornare l'elemento in posizione 1 del vettore v.

```
int heapMin(heap H){
    return H.v[1];
}
```

Nell'implementazione dell'operatore **heapinsert** il nodo u che viene fatto risalire fino, eventualmente, alla radice è semplicemente un indice del vettore. Di volta in volta viene confrontato il suo peso con quello del padre che si trova in posizione $u/2$. Viene restituito l'heap modificato.

```
heap heapInsert(heap H, int e){
    int u;
    if(H.n >= c)
        return H;
    H.n++;
    H.v[H.n] = e;
    u = H.n;
    while(u > 1 && H.v[u] < H.v[u/2]){
        swap(&(H.v[u/2]), &(H.v[u]));
        u = u/2;
    }
    return H;
}
```

La funzione non presenta particolari degni di nota tranne che prima di aggiungere un elemento si verifica se c'è spazio disponibile sul vettore. Infine lo scambio dei due pesi avviene attraverso una funzione che prende in input gli indirizzi delle variabili contenenti gli interi da scambiare. Ecco come è implementata.

```
void swap(int *a, int *b){
    int t;
    t = *a;
    *a = *b;
    *b = t;
}
```

Per ultima consideriamo la funzione che cancella l'elemento di costo minimo dall'heap.

```
heap heapDel(heap H){
    int u;
    if (H.n == 0)
        return H;
    H.v[1] = H.v[H.n];
    H.n--;
    u = 1;
    while( 2*u+1 <= H.n && (H.v[u] > H.v[2*u] || \
                            H.v[u] > H.v[2*u+1])){
        if(H.v[2*u] < H.v[2*u + 1]){
            swap(&(H.v[u]), &(H.v[2*u]));
            u = 2*u;
        } else {
            swap(&(H.v[u]), &(H.v[2*u + 1]));
            u = 2*u + 1;
        }
    }
    if(2*u <= H.n && H.v[u] > H.v[2*u]){
        swap(&(H.v[u]), &(H.v[2*u]));
        u = 2*u;
    }
    return H;
}
```

Dopo aver verificato che l'heap contiene almeno un elemento, il minimo nella prima posizione del vettore viene sostituito con l'ultimo elemento che rappresenta la foglia dell'ultimo livello più a destra. Inizia la discesa del nodo u verso la foglia. Questa va avanti fintanto che u ha entrambi i figli e almeno uno di questi ha un peso inferiore di quello di u. Arrivato alla fine del ciclo può succedere che u abbia un solo figlio (quindi in posizione $2u$). Non avendo altri figli allora u si trova al penultimo livello dell'albero. In tal caso se il peso di questo è inferiore a quello di u avviene lo scambio.

L'implementazione appena mostrata ci permette di applicare i Teoremi 7.1 e 7.2 e affermare che le operazioni di inserimento e cancellazione del minimo hanno costo logaritmico nel numero dei nodi. Invece l'operazione di lettura del minimo ha costo costante.

7.3 Applicazione all'ordinamento di un vettore

Si supponga di dover ordinare un vettore x di n elementi interi. Se gli elementi del vettore sono organizzati in un heap H, il vettore può essere ordinato estraendo di volta in volta il minimo da H che andrà inserito in posizione via via crescente nel vettore x. Ogni estrazione di minimo costa al più $\Theta(\log n)$ che viene ripetuta n volte. Inoltre con costo $\Theta(n \log n)$ gli elementi del vettore possono essere inseriti all'interno di H inizialmente vuoto. Quello che abbiamo appena descritto è un algoritmo di ordinamento di complessità ottimale (ovvero $\Theta(n \log n)$) chiamato *heap-sort*. Ecco il codice C.

```
void heapsort(int x[], int m){
     heap H = heapNew();
     int i;

     for(i = 0; i < m; i++)
          H = heapInsert(H, x[i]);

     i = 0;
     while(H.n > 0){
          x[i] = heapMin(H);
          H = heapDel(H);
          i++;
     }
}
```

7.4 Vettori di dimensione variabile

Il vero punto debole nell'implementione dell'heap con un vettore sta nella capacità del vettore che deve essere decisa a priori. La capacità deve essere scelta in modo che non si corra il rischio di usare tutto lo spazio riservato quindi deve essere una sovrastima dell'effettivo utilizzo del vettore. Tuttavia si corre il rischio di sovrastimare troppo e sprecare troppa memoria. Infine, anche quando la capacità viene stimata correttamente può succedere che la coda abbia una grande variabilità di dimensioni e quindi la maggior parte del tempo questa potrebbe essere sotto-utilizzata. D'altra parte il vettore è comodo in quanto permette l'accesso in tempo costante ad un nodo dell'albero conoscendone il padre o uno dei figli. Questo rappresenta un grosso vantaggio che non viene offerto dalle strutture dinamiche. In questo paragrafo mostreremo una soluzione che cerca di sintetizzare i vantaggi delle strutture dinamiche con quelli del vettore.

L'idea è quella di utilizzare un vettore di una certa capacità iniziale, quando questo risulta essere pieno viene sostituito da un altro più grande sul quale verranno copiati tutti gli elementi del primo. Quando questo risulta essere troppo sotto-utilizzato verrà rimpiazzato da uno più piccolo. Il ridimensionamento del vettore è una operazione che comporta, oltre la creazione di un nuovo vettore, anche la copia di tutti gli elementi dal vecchio vettore. Inutile dirlo, questa è una operazione molto costosa ($\Theta(n)$ se n è il numero di elementi nel vettore) che non ci possiamo permettere ad ogni inserimento o cancellazione. Però se avessimo la garanzia che questa operazione venga eseguita almeno dopo n passi (ognuno dei quali richiede tempo costante) allora il costo dell'operazione di ridimensionamento, se viene visto come spalmato tra gli $\Omega(n)$ passi che separano due ridimensionamenti successivi, ha costo virtualmente costante. Questo modo di misurare la complessità degli algoritmi considerando la media dei costi di una sequenza di operazioni è detta *complessità ammortizzata*.

Per i vettori a dimensione variabile definiremo una nuova struttura dati chiamata *vector* che può essere utilizzata come un array con in più la possibilità di aggiungere o eliminare un elemento del vettore. In particolare sarà definita l'operazione `vectorInsert` che inserisce un nuovo elemento nell'ultima posizione del vector e `vectorDelete` che elimina l'ultimo elemento dal vettore.

Ad ogni vector sono associati due interi: una capacità c che indica la dimensione del vettore ed n che indica il numero di elementi presenti nel vettore. Se il vettore ha almeno una posizione libera ($n < c$), l'operazione `vectorInsert` inserisce un nuovo elemento

come ultimo elemento del vector. Altrimenti, se il vettore è pieno ($n = c$) allora viene creato un nuovo vettore di dimensione $c' = 2(n + 1)$, tutti gli elementi del vecchio vettore vengono copiati nel nuovo e, infine, viene aggiunto il nuovo elemento come ultimo elemento del vettore creato. Viceversa, l'operazione `vectorDelete`, elimina l'ultimo elemento del vettore, ovvero n viene decrementato di uno facendo si che l'ultimo elemento venga ignorato. Se n è uguale a $c/4$ ($c = 4n$), viene creato un nuovo vettore di dimensione $c' = c/2$ (ovvero $c' = 2n$) che andrà a contenere tutti gli elementi del vector. Questa operazione ha lo scopo di rendere limitata la quantità di memoria non utilizzata.

Teorema 7.3. *In media le operazioni* `vectorInsert` *e* `vectorDelete` *hanno costo computazionale costante. Inoltre il numero di elementi nel vettore è almeno un quarto della sua capacità.*

Dimostrazione. Dopo una operazione di inserimento o cancellazione (chiamiamola o_1) che modifica la dimensione c del vettore si ha che la capacità del vettore $c = 2n$. Supponiamo che la prossima operazione che causa il ridimensionamento (chiamiamola o_2) sia una insert allora se n' è il numero di elementi nel vettore deve essere $n' = c$. Quindi sono state eseguite almeno n operazioni di costo costante (nel caso peggiore n insert). Il costo dell'ultima insert sarà $\Theta(n') = \Theta(n)$ che se distribuito sulle n operazioni che l'hanno preceduta comporta un costo medio costante. Tra l'operazione o_1 ed o_2 il numero di elementi nel vettore n' non è mai stato uguale a $c/4 = n/2$ altrimenti sarebbe stata invocata una operazione delete che avrebbe ridimensionato il vettore quindi $n' > c/4$.

Se o_2 è una operazione di delete allora $n' = c/4 = n/2$. Quindi, o_2 viene eseguita almeno dopo $n/2 = n'$ passi di costo costante (n' operazioni di delete) mentre l'operazione o_2 ha un costo $\Theta(n') = \Theta(n)$. Anche in questo caso, ad ogni passo il numero di elementi nel vettore è sempre almeno $c/4$. □

7.4.1 Implementazione in C

Un vector è definito da un vettore contente elementi del tipo desiderato (nel nostro caso `int`), da una capacità del vettore e da un intero che ne rappresenta l'effettivo utilizzo. Queste tre entità costituiranno la `struct vector`.

```
struct vector{
  int *array;
  int c;
  int n;
};
typedef struct vector vector;
```

Si osservi che il campo `array`, che rappresenta il vettore di dimensione c, è definito soltanto come un puntatore a `int`. Questo perché la sua dimensione è ancora indefinita. Questa definizione non contrasta con l'altro modo di definire vettori per esempio

```
int v[20];
```

in quanto, come è stato detto, il valore di v è l'indirizzo di memoria (o puntatore) del primo elemento del vettore di 20 elementi che viene creato contestualmente alla definizione della variabile v. Mentre definendo v in questo modo

```
int *v;
```

stiamo dicendo che *v* conterrà l'indirizzo di memoria di un intero che ancora non viene definito. Potremmo assegnare un valore a *v* utilizzando la funzione `malloc`.

```
v = malloc(sizeof(int)*20);
```

L'effetto di questa istruzione è allocare un numero di celle di memoria consecutive sufficienti a contenere 20 interi. L'indirizzo del primo intero viene assegnato a *v*. Anche in questo caso, *v* conterrà l'indirizzo del primo intero di una sequenza di 20 esattamente come nella definizione esplicita del vettore *v* di 20 elementi.

Ritornando alla definizione del vector, risulta utile una funzione che inizializza la struttura allocando memoria al campo array.

```
vector newvector(int c){
  vector v;
  v.array = (int*)malloc(sizeof(int)*c);
  v.c = c;
  v.n = 0;
  return v;
}
```

La funzione `newvector` non fa altro che creare un array di *c* elementi.

Ora è possibile definire le funzioni `vectorInsert` e `vectorDelete`.

```
vector insert(vector v, int e){
  if(v.n<v.c){
    v.array[v.n]=e;
    v.n++;
    return v;
  } else {
    vector new_v;
    int i;
    new_v = newvector(2*v.c+1);
    for(i=0; i<v.n; i++)
      new_v.array[i] = v.array[i];
    new_v.array[v.n]=e;
    new_v.c = 2*v.c;
    new_v.n = v.n+1;
    return new_v;
  }
}
```

```
vector delete(vector v){
  v.n--;
  if(v.n>(v.c)/4){
    return v;
  } else {
    vector new_v;
    int i;
    new_v = newvector((v.c)/2);
    for(i=0; i<v.n; i++)
      new_v.array[i] = v.array[i];
    new_v.c = (v.c)/2;
    new_v.n = v.n;
    return new_v;
  }
}
```

Le due funzioni non presentano particolari innovazioni. Nel caso in cui l'effetto dell'inserimento o dell'cancellazione non altera i vincoli imposti tra la i valori *c* ed *n*, viene modificato il vector *v* in input in modo appropriato, altrimenti viene creato – nel rispetto dei vincoli – il vector `new_v` che viene restituito in output.

Per ora ci siamo limitati a definire un vector generico. Termineremo il capitolo mostrando come possa essere utilizzato nell'implementazione di un heap.

Innanzi tutto definiamo un heap come un vector.

```
typedef vector heap;
```

Un nuovo heap viene creato con la seguente funzione che crea un vector di capacità 10 e imposta *n* a 1 in quanto il primo elemento del vettore, ovvero quello in posizione 0, viene ignorato.

```
heap heapNew(){
    heap H;
    H = newvector(10);
    H.n = 1; /* saltiamo la posizione 0 */
    return H;
}
```

Infine le funzioni di inserimento e cancellazione utilizzano le funzioni di inserimento e cancellazione sul vector.

```
heap heapInsert(heap H, int e){
    int u;
    H = insert(H, e);
    u = H.n - 1;
    while(u > 1 && H.array[u] < H.array[u/2]){
        swap(&(H.array[u/2]), &(H.array[u]));
        u = u/2;
    }
    return H;
}
```

```
heap heapDel(heap H){
    int u;
    H.array[1] = H.array[H.n-1];
    H = delete(H);
    u = 1;
    while( 2*u+1 < H.n && \
            (H.array[u] > H.array[2*u] || \
            H.array[u] > H.array[2*u+1])){
        if(H.array[2*u] < H.array[2*u + 1]){
            swap(&(H.array[u]), &(H.array[2*u]));
            u = 2*u;
        } else {
            swap(&(H.array[u]), \
                &(H.array[2*u + 1]));
            u = 2*u + 1;
        }
    }
    if(2*u < H.n && H.array[u] > H.array[2*u]){
        swap(&(H.array[u]), &(H.array[2*u]));
        u = 2*u;
    }
    return H;
}
```

Nel caso peggiore la complessità delle funzioni **heapinsert** e **heapdel** è $\Theta(n + \log n)$ (dove n è la dimensione della struttura) dovuto al ridimensionamento ($\Theta(n)$) e il ripristino delle proprietà dell'heap ($\Theta(\log n)$). Tuttavia il ridimensionamento avviene ogni n operazioni di costo $\Theta(\log n)$ (gli inserimenti e cancellazioni che non prevedono il ridimensionamento). Quindi il costo delle almeno n operazioni che separano due inserimenti o cancellazioni è

$$(n + \log n) + n \log n$$

che spalmato tra le n operazioni comporta un costo medio $\Theta(\log n)$ per ogni operazione.

Capitolo 8

Alberi binari di ricerca

La struttura dati heap ci permette di organizzare le informazioni in modo che risultino computazionalmente efficienti le operazioni di ricerca del minimo, cancellazione del minimo ed inserimento di un nuovo elemento. In particolare, se n è il numero di elementi che fanno parte della collezione, le su citate operazioni hanno una implementazione di costo $\Theta(\log n)$.

Tuttavia altre operazioni come la ricerca oppure la cancellazione di un elemento qualsiasi ha un costo computazionale più elevato ($\Omega(n)$). In questo capitolo introdurremo una struttura dati sulla quale anche queste ultime operazioni saranno implementabili in modo efficiente.

8.1 Vettori ordinati e ricerca binaria

Sia v un vettore di n interi ordinati in modo crescente. Vogliamo un algoritmo efficiente che stabilisca se un intero x si trova in v o meno. Un algoritmo che va bene in tutti i casi scorre il vettore dal primo all'ultimo elemento verificando che l'elemento analizzato sia x. In caso affermativo l'algoritmo termina con esito positivo altrimenti si va avanti fino alla fine del vettore. Il costo di questo algoritmo è lineare nella dimensione del vettore. Un altro algoritmo più furbo tiene conto dell'ordinamento: verifica se $v[n/2]$ è x, in caso affermativo l'algoritmo termina altrimenti se $v[n/2] > x$ ripete l'algoritmo sulla metà destra del vettore v altrimenti su quella sinistra. L'algoritmo, chiamato *ricerca binaria*, può essere descritto ricorsivamente nel modo che segue.

binsearch(x, v, l, r)

1. se $l > r$ concludi che $x \notin v$ altrimenti
   (a) $m = (l + r)/2$
   (b) se $v[m] = x$ concludi che $x \in v$ altrimenti
      i. se $v[m] < x$ esegui **binsearch**$(x, v, m + 1, r)$ altrimeni esegui **binsearch**$(s, v, l, m - 1)$

L'algoritmo cerca x nel sotto-vettore delimitato dagli indici l ed r con $l \leq r$. Verifica se x si trova nella posizione mediana m del sotto-vettore. Se così non è sposta la ricerca sul sotto-vettore a sinistra o a destra di m a secondo del valore di x rispetto $v[m]$. La ricerca prosegue fino a quando il vettore delimitato da l ed r è non vuoto. L'algoritmo deve essere invocato su tutto il vettore ovvero nel seguente modo **binsearch**$(x, v, 0, n - 1)$.

Teorema 8.1. *L'algoritmo* **binsearch** *conclude correttamente la ricerca di un intero x fra gli n di un vettore ordinato v in tempo, nel caso peggiore, $\Theta(\log n)$.*

Dimostrazione. La correttezza segue dal fatto che ogni volta l'algoritmo elimina dalla ricerca la parte del vettore in cui x non può essere presente e si concentra sull'altra. Se $T(n)$ indica il tempo di esecuzione dell'algoritmo su una sequenza di n elementi allora se c e d sono costanti, $T(1) = d$ altrimenti per $n > 1$, $T(n) = T(n/2) + c$. Ovvero

$$T(n) = T(n/2) + c = T(n/4) + 2c = ... = T(n/2^k) + kc.$$

Per $k \approx \log n$, $T(n) = d + c \log n \in \Theta(\log n)$. □

L'implementazione in C dell'algoritmo segue pedissequamente la descrizione fatta precedentemente.

```
int binsearch(int x, int v[], int l, int r){
    int m;
    if(l > r)
        return -1;
    m = (l+r)/2;
    if(v[m] == x)
        return m;
    if(x < v[m])
        return binsearch(x, v, l, m-1);
    return binsearch(x, v, m+1, r);
}
```

Esso restituisce l'indice del vettore in cui si trova x nel caso in cui questo venga trovato, altrimenti restituisce -1.

Con la stessa tecnica utilizzata nel Capitolo 4 per calcolare il numero di confronti necessari per ordinare un vettore (Teorema 4.2) è possibile dimostrare che il numero di confronti richiesto (nel caso peggiore) per ricercare un elemento in una sequenza di dimensine n è logaritmico nella dimensione della sequenza. La differenza tra le due dimostrazioni sta nel fatto che per l'ordinamento la dimensione dello spazio delle soluzioni è $n!$ (ogni possibile permutazione degli elementi è una soluzione ammissibile) mentre per la ricerca questo è $n + 1$, infatti l'elemento cercato può essere in una delle n posizioni del vettore oppure può non esser presente. Ci si riferisca alla dimostrazione del Teorema 4.2, se ℓ è il numero di confronti necessario al raggiungimento di una soluzione allora $2^\ell \geq n+1$ ovvero $\ell \in \Omega(\log n)$. Concludiamo che vale il seguente risultato.

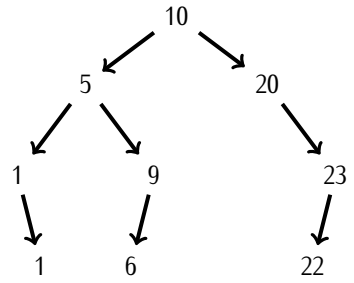
Teorema 8.2. *Ogni algoritmo di ricerca (corretto) basato su confronti richiede, nel caso peggiore, tempo* $\Omega(\log n)$ *dove* n *è la dimensione dell'insieme in cui ricercare l'elemento.*

Quindi l'algoritmo **binsearch** è ottimo. Tuttavia il vettore ordinato non è utilizzabile per rappresentare collezioni di insiemi in cui sono previste operazioni di inserimento e cancellazione di elementi. Infatti queste operazioni richiedono costo elevato (almeno lineare) per riaggiustare la struttura ovvero per mantenere ordinato l'insieme.

8.2 Alberi binari di ricerca

Diciamo subito che la struttura che presenteremo in questa sezione non garantisce buone prestazioni per le operazioni di ricerca, inserimento e cancellazione. Tuttavia, come vedremo, se opportunamente affinata, ci permetterà di raggiungere il nostro scopo finale ovvero una struttura per la quale tutte le operazioni hanno un costo logaritmico.

Come già detto, un albero con radice è binario se ogni nodo ha al più due figli e questi sono ordinati e quindi è possibile distinguere il figlio destro ed il figlio sinistro di ogni nodo. Un *albero binario di ricerca* è un albero binario in cui ogni nodo u contiene una informazione (o *peso*) numerica[1] $i(u)$. Inoltre per ogni nodo u, $i(u)$ deve essere maggiore del peso contenuto in tutti i nodi nel suo sotto-albero sinistro (il sotto-albero che ha per radice il figlio sinistro di u) e minore o uguale delle informazioni contenute in tutti i nodi nel suo sotto-albero destro. Per formalizzare meglio questo concetto e quelli a venire diamo qualche altra definizione: se u è un nodo dell'albero binario T con T_u indichiamo il sotto-albero di T che ha per radice u inoltre con $s(u)$ e $d(u)$ indichiamo rispettivamente il figlio sinistro e destro di u. Quindi per ogni nodo u in un albero binario di ricerca T deve valere: per ogni nodo $v \in T_{s(u)}$, $i(v) < i(u)$ ed inoltre per ogni nodo $v \in T_{d(u)}$, $i(v) \geq i(u)$. In figura è mostrato un esempio di albero binario di ricerca.



Per ricercare una chiave x nell'albero si parte dalla radice r se $x = i(r)$ allora il nodo è stato trovato altrimenti se $x < i(r)$ per forza di cose il nodo deve essere ricercato nel sotto-albero sinistro di r altrimenti nel sotto-albero destro. Quindi nel primo caso la ricerca continua ricorsivamente su $T_{s(r)}$, nel secondo su $T_{d(r)}$. Se la ricerca porta verso un nodo non esistente si conclude che la chiave x non esiste. Per esempio la ricerca di un nodo con chiave 9 nell'albero in figura comporterà i seguenti passi:

- confronta 10 con 9, $9 < 10$ quindi continua la ricerca sul sotto-albero con radice 5;
- confronta 9 con 5, $9 \geq 5$ quindi continua la ricerca sul sotto-albero con radice 9
- il nodo è trovato.

Invece la ricerca di un nodo contenente la chiave 15 comporterebbe la ricerca di questa nel sotto-albero sinistro del nodo contenente la chiave 20 (tale nodo si dovrebbe trovare nel sotto-albero destro della radice e nel sotto-albero sinistro del nodo con chiave 20), questo sotto-albero è vuoto quindi 15 non appartiene alla struttura.

Prima di proseguire diamo una descrizione formale dell'algoritmo di ricerca chiamato **bintreesearch**. Useremo la notazione $r(T)$ per indicare il nodo radice dell'albero T.

bintreesearch(T, x)

1. Se $T = \emptyset$, restituisci $\emptyset$.
2. Sia $r = r(T)$
3. Se $x = i(r)$, il nodo è stato trovato, restituisci r.

[1] Come al solito il fatto che sia strettamente numerica non ha importanza, quello che importa è che esista un ordinamento totale tra le informazioni contenute nei nodi. Senza perdita di generalità assumeremo che l'informazione sia di tipo intero.

4. Se $x < i(r)$, esegui **bintreesearch**$(T_{s(r)}, x)$ altrimenti
   esegui **bintreesearch**$(T_{d(r)}, x)$.

L'algoritmo è descritto con una funzione ricorsiva che ricerca la chiave x nell'albero T. La ricerca continua nel sotto-albero sinistro o destro della radice di T secondo che x sia minore o maggiore-uguale della chiave contenuta nella radice di T.

L'inserimento di un nuovo nodo avente chiave x procede in modo simile alla ricerca. Il nuovo nodo sarà una foglia dell'albero modificato, la ricerca individuerà la posizione in cui si troverebbe il nodo se questo facesse parte dell'albero. Si supponga di dover aggiungere un nodo con etichetta 15 nell'albero dell'esempio mostrato precedentemente: confrontando l'etichetta con il peso della radice 10 deduciamo che il nuovo nodo deve essere inserito nel sotto-albero destro della radice; poiché $15 < 20$ il nuovo nodo deve essere inserito nel sotto-albero sinistro del nodo con etichetta 20; questo nodo non ha un sotto-albero sinistro quindi il nuovo nodo verrà aggiunto come figlio sinistro di questo.

bintreeinsert(T, x)

1. Se $T = \emptyset$, crea la radice di T con etichetta x .
2. Sia $u = r(T)$
3. Fintanto che il nodo non è inserito
   (a) se $x < i(u)$
      i. se u non ha figlio sinistro inserisci un nuovo nodo con peso x come figlio sinistro di u
      altrimenti $u = s(u)$
   (b) altrimenti
      i. se u non ha figlio destro inserisci un nuovo nodo con peso x come figlio destro di u
      altrimenti $u = d(u)$
4. restituisci T

Ora consideriamo l'operazione **bintreemin**, ovvero la ricerca del nodo contenente il peso minimo. Come al solito si parte dalla radice r dell'albero T. Poiché tutti i nodi nel sotto-albero sinistro di r hanno peso minore di quello in r e tutti quelli sul sotto-albero destro hanno peso maggiore o uguale di quello in r, il minimo dell'albero va ricercato sul sotto-albero sinistro di r. Se r non ha figlio sinistro allora il minimo si trova proprio in r. Da questa osservazione ricaviamo un semplice algoritmo per la ricerca del minimo: si parte dalla radice e si avanza di figlio sinistro in figlio sinistro, quando si giunge ad un nodo che ne è privo quel nodo contiene il peso minimo.

bintreemin(T)

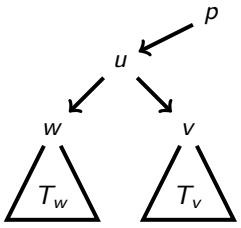
1. $u = r(T)$
2. fintanto che u ha figlio sinistro
   (a) $u = s(u)$;
3. restituisci u.

L'ultima operazione che considereremo è **bintreedelete** che ha la finalità di eliminare dall'albero binario di ricerca T un nodo avente peso x indicato come input della funzione. Per prima cosa raggiungiamo il nodo u da eliminare utilizzando l'operazione di ricerca. Nel

caso in cui u sia una foglia può essere eliminato dall'albero senza problemi. Nel caso in cui abbia soltanto un figlio v si può eliminare u facendo in modo che v diventi figlio del padre p di u. Si consideri la situazione mostrata in figura: u è figlio sinistro di p e v è figlio destro di u.

Si può eliminare u sostituendolo col suo unico figlio mantenendo inalterate le proprietà dell'albero binario di ricerca.

La situazione diventa più complicata quando il nodo da cancellare ha entrambi i figli. In figura è rappresentato il nodo da cancellare u, il padre p ed sotto-alberi sinistro e destro di u con radici, rispettivamente, w e v.

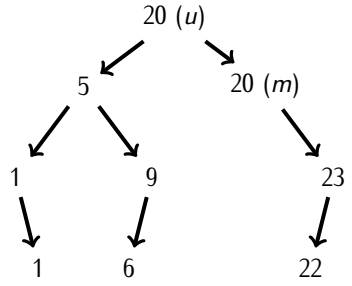


Cominciamo con l'osservare che possiamo rimpiazzare il peso di u con il peso m minimo in T_v in quanto $m \geq i(u) > i(x)$ per ogni $x \in T_w$ e $m \leq i(x)$ per ogni $x \in T_v$. Ora il problema è eliminare il nodo di peso minimo in T_v. Questa è una operazione che sappiamo eseguire in quanto il nodo di peso minimo non ha figlio sinistro (se lo avesse sarebbe quest'ultimo il minimo).

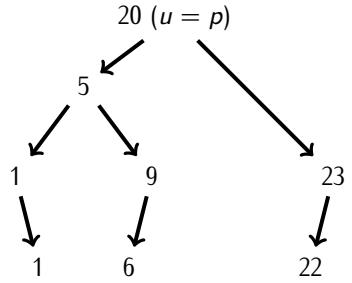
bintreedelete(T, x)

1. $u =$ **bintreesearch**(T, x), sia p il padre di u;
2. se $u = \emptyset$, il nodo non esiste, fine;
3. se u è una foglia eliminalo da T, fine;
4. se u ha solo un figlio v, elimina u e rendi v figlio di p al posto di u; se u è la radice, rendi v la nuova radice, fine;
5. sia v il figlio destro di u;
6. $m =$ **bintreemin**(T_v), sia q il padre di m;
7. $i(u) = i(m)$;
8. se m è una foglia eliminalo da T, fine;
9. elimina m e rendi il figlio destro di m figlio di q al posto di m.

Come esempio eliminiamo il nodo di peso 10 (la radice) dall'albero a pagina 82. Il nodo u (ovvero la radice) ha entrambi i figli, quindi eseguiamo la ricerca del minimo sul sotto-albero con radice 20; il minimo è proprio 20 quindi m è il figlio destro di u; il nuovo peso di u diventa 20.



Il padre q di m è lo stesso nodo u. Con l'ultimo passo rendiamo il figlio destro di m figlio di q al posto di m.



La complessità degli operatori che abbiamo appena definito dipende dall'altezza h dell'albero. Supponiamo che in tempo costante possiamo accedere alla radice dell'albero e, dato un nodo dell'albero, ancora in tempo costante possiamo accedere ai suoi figli ed all'informazione ivi presente. Il numero di operazioni eseguite dalla funzione **bintreesearch** su un albero di altezza h è

$$C(h) = \begin{cases} c & \text{se } h = 0 \\ d + C(h-1) & \text{se } h \geq 1 \end{cases}$$

dove c e d sono costanti. Se $h = 0$ l'algoritmo termina al primo passo altrimenti, dopo alcune operazioni di costo costante, viene invocata la funzione su un albero di altezza $h-1$. Il caso peggiore accade quando si percorre il cammino radice-foglia più lungo (ovvero quello lungo h) quindi, risolvendo l'equazione di ricorrenza, si ottiene $C(h) = hd + c$ ovvero l'algoritmo è lineare dell'altezza dell'albero. La funzione **bintreeinsert** non è altro una ricerca del padre del nuovo nodo lungo un percorso radice-foglia. L'inserimento vero e proprio avviene in tempo costante quindi anche questa funzione costa, nel caso peggiore $\Theta(h)$. La funzione **bintreemin**, nel caso in cui il minimo sia la foglia più lontana dalla radice, costa $\Theta(h)$. Infine l'operazione **bintreedelete** esegue, nel caso peggiore, una ricerca ed una ricerca di un minimo. Il costo complessivo è $\Theta(h)$.

8.2.1 Implementazione in C

Per implementare un nodo dell'albero binario (chiamiamolo `Bnode`) utilizzeremo le stesse idee applicate per le liste. Ogni nodo è definito da una `struct` di tre campi: un campo per

l'informazione (nel nostro caso un peso intero), un riferimento all'eventuale figlio sinistro e un riferimento all'eventuale figlio destro. Un albero binario (`bintree`) è identificato dal riferimento alla radice dalla quale è possibile raggiungere, attraverso i figli, ogni altro nodo dell'albero.

```
struct bnode {
    int info;
    struct bnode *left;
    struct bnode *right;
};
typedef struct bnode Bnode;
typedef Bnode *bintree;
```

I campi `left` e `right` della struttura sono i puntatori ai figli sinistro e destro del nodo descritto dalla struttura. Alternativamente si possono vedere come riferimenti ai sotto-alberi sinistro e destro del nodo.

La funzione **bintreesearch** viene implementata come descritto precedentemente da una funzione ricorsiva.

```
Bnode *bintreeSearch(bintree T, int x){
    if(T==NULL)
        return NULL;
    if(T->info == x)
        return T;
    if(x < T->info)
        return bintreeSearch(T->left, x);
    else
        return bintreeSearch(T->right, x);
}
```

La ricerca parte dalla radice dell'albero e, in caso di insuccesso, prosegue sul sotto-albero sinistro o destro secondo che la chiave cercata sia minore o maggiore-uguale di quella contenuta nella radice. Viene restituito il puntatore al nodo contenente l'informazione cercata. Se questo non viene trovato viene restituito `NULL`. Nel caso in cui ci siano più nodi contenenti la chiave cercata viene restituito il primo nodo trovato. Osserviamo che possiamo applicare a questa implementazione tutte le considerazioni fatte precedentemente sulla complessità dell'algoritmo di ricerca. Concludiamo quindi che questa implementazione della funzione **bintreesearch** ha complessità, nel caso peggiore, $\Theta(h)$ dove h è l'altezza dell'albero.

La funzione **bintreeinsert** che aggiunge un nuovo nodo di con etichetta x in un albero T è implementata in C dalla funzione `bintreeAddNewNode`.

```
Bnode *bintreeAddNewNode(bintree T, int x){
    Bnode *u;
    int finito = 0;
    if(T == NULL)
        return bintreeNewNode(x);
    u = T;
    while(finito==0){
        if(x < u->info){
            if(u->left == NULL){
                u->left = bintreeNewNode(x);
                finito = 1;
            } else
                u = u->left;
        } else {
            if(u->right == NULL){
                u->right = bintreeNewNode(x);
                finito = 1;
            } else
                u = u->right;
        }
    }
    return T;
}
```

Se l'albero è vuoto viene creato un nuovo nodo che diventerà la radice dell'albero. Verrà restituito l'indirizzo di questo nodo. Altrimenti inizia la ricerca del futuro padre del nuovo nodo: alla variabile `finito` verrà assegnato il valore 1 quando questo nodo sarà trovato. Il padre p del nuovo nodo sarà quello avente peso maggiore del peso di x e privo di figlio sinistro (cosicché il nodo inserito sarà figlio sinistro p) oppure quello avente peso minore-uguale e privo di figlio destro (quindi il nodo inserito sarà figlio destro di p). La creazione del nuovo nodo è affidata alla funzione `bintreeNewNode` che restituisce l'indirizzo del nodo creato.

```
Bnode *bintreeNewNode(int x){
        Bnode *u = malloc(sizeof(Bnode));
        u->info = x;
        u->left = NULL;
        u->right = NULL;
        return u;
}
```

Si osservi che questa funzione ha costo costante quindi la funzione `bintreeAddNewNode` ha costo al più lineare nell'altezza dell'albero.

L'implementazione della funzione **bintreemin** è immediata e segue dalla definizione dell'algoritmo.

```
Bnode *bintreeMin(bintree T){
    Bnode *u;
    if(T == NULL)
        return NULL;
    u = T;
    while(u->left != NULL)
        u = u->left;
    return u;
}
```

Se h è l'altezza dell'albero T, la complessità di questa funzione è $\Theta(h)$.

La funzione **bintreedelete** è leggermente più macchinosa e quindi anche la sua implementazione ne risente. Innanzi tutto iniziamo col definire una funzione che elimina un nodo u. La funzione ha per input il puntatore ad u, il puntatore p al padre di u ed un intero lr inoltre sappiamo che u ha al più un figlio. L'intero lr serve a specificare se u è figlio sinistro o destro di p, in particolare se u è figlio sinistro allora $lr = -1$, se u è figlio destro $lr = 1$ mentre se u non ha padre (è la radice dell'albero) $lr = 0$. In questo ultimo caso $p =$ NULL. La funzione restituisce p nel caso in cui u non è la radice di T, altrimenti restituisce la nuova radice dell'albero.

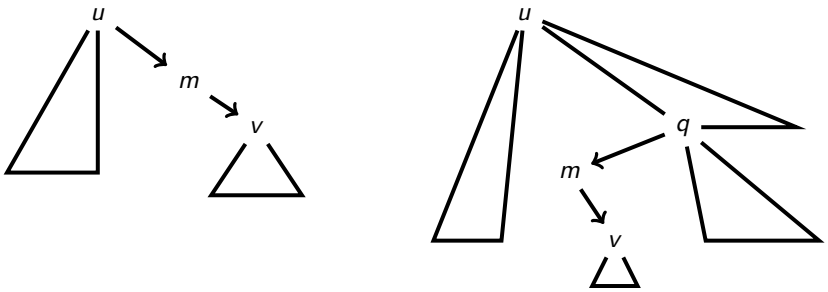
```
Bnode *bintreeDL(Bnode *u, Bnode *p, int lr){
    Bnode *v;
    if(u->left == NULL && u->right == NULL){
        /* u e' una foglia */
        free(u);
        if(lr==-1)
            p->left=NULL;
        else if (lr==1)
                p->right=NULL;
        else p = NULL; /* u era la radice */
        return p;
    }
    /* u ha solo un figlio */
    if(u->left != NULL)
        v = u->left;
    else
        v = u->right;
    free(u);
    if(lr == -1)
        p->left = v;
    else if(lr == 1)
        p->right = v;
    else
        p =v; /* u era la radice */

    return p;
}
```

Questa funzione viene invocata all'interno della funzione principale solo nel caso in cui u ha al più un figlio quindi assumiamo che questa proprietà valga. Nel caso in cui non ha

figli, ovvero u è una foglia, u può essere eliminato direttamente. Occorre tuttavia eliminare ogni traccia di u dal nodo padre. Quindi se $lr = -1$ allora u era figlio sinistro di p e quindi il nuovo valore del puntatore `left` di p deve essere `NULL`. Qualcosa di analogo avviene se $lr = 1$. Se invece $lr = 0$ allora u era la radice dell'albero. Dal fatto che questa fosse senza figli deduciamo che l'albero era composto dal solo nodo u, eliminando questo nodo resta l'albero vuoto e quindi viene restituito `NULL`. Se u ha un (unico) figlio lo facciamo puntare da v, eliminiamo u e v prenderà il suo posto. Se u era figlio sinistro di p ($lr = -1$) allora v diventerà il nuovo figlio sinistro di p, se u era figlio destro ($lr = 1$), v diventerà nuovo figlio destro di p altrimenti u era la radice dell'albero. In questo ultimo caso v sarà la nuova radice dell'albero. Come utilizzare l'output dei questa funzione sarà chiaro una volta definita l'implementazione della funzione **bintreedelete**. Questa funzione elimina dall'albero T un nodo di peso x. Se tale nodo non esiste la funzione restituisce l'albero inalterato.

```
bintree bintreeDelete(bintree T, int x){
    Bnode *u,*p,*v,*q,*m;
    int lr = 0; /* lr = -1 se u figlio sinistro di p
                   lr  = +1 se figlio destro*/
    /* ricerca del nodo da eliminare e del padre */
    u = T;
    while(u!=NULL && u->info!=x){
        p = u;
        if(x < u->info){
            u = u->left;
            lr = -1;
        } else {
            u = u->right;
            lr = 1;
        }
    }
    if(u==NULL)
        return T;
    /* se u ha al piu' un figlio */
    if(u->left == NULL || u->right == NULL){
        if(p==NULL)
            return bintreeDL(u, p, lr);
        else {
            p = bintreeDL(u, p, lr);
            return T;
        }
    }
    /* u ha due figli */
    /* cerchiamo il minimo */
    m = u->right;
    q = NULL;
    while(m->left != NULL){
        q = m;
        m = m->left;
    }
    u->info = m->info;
    if(m->right == NULL){ /*m e' una foglia*/
        free(m);
        v=NULL;
    } else {
        v = m->right;
        free(m);
    }
    if(q==NULL) /* il padre di m era u */
        u->right = v;
    else{
        q->left = v;
    }
    return T;
}
```

Per prima cosa la funzione cerca un nodo avente peso x. Qui la procedura di ricerca viene implementata di nuovo in quanto insieme al riferimento al nodo da eliminare u ci occorre anche il riferimento al padre p. Inoltre abbiamo bisogno di sapere se u è figlio sinistro o destro di p, per questo facciamo uso di una variabile intera lr che ha valori in $\{-1, 1, 0\}$ secondo che u sia figlio sinistro o destro di p oppure che u sia radice. Nel caso in cui u sia privo di uno dei due figli applicheremo la funzione `bintreeDL` definita precedentemente. In particolare se u è la radice dell'albero restituiamo il risultato della funzione `bintreeDL` che nominerà una nuova radice, altrimenti questa funzione troverà un nuovo figlio per p al posto di u quindi restituiamo il riferiemnto all'albero T. Se invece u ha entrambi i figli cerchiamo il minimo del sotto-albero destro di u che chiamiamo m. Ancora una volta non utilizziamo l'implementazione della funzione **bintreemin** in quanto oltre al nodo m contenente il minimo abbiamo anche bisogno del riferimento al padre (diciamo q). Dopo aver copiato il campo `info` di m in u passiamo alla eliminazione di m che verrà sostituito dal figlio puntato da v. Per prima cosa osserviamo che m non può avere figlio sinistro altrimenti questo avrebbe peso minore di quello di m. Se m non ha neanche il figlio destro allora v è `NULL` altrimenti a v viene assegnato l'indirizzo del figlio destro di m. Ora v deve diventare il nuovo figlio del padre di m al posto dello stesso m. Abbiamo due possibilità: m era figlio di u (figura a sinistra) oppure m figlio di un nodo q diverso da u (figura a destra).



Nel primo caso (segnalato dal fatto che q ha mantenuto il valore `NULL`) v diventeà figlio destro di u. Nel secondo caso v diventerà figlio sinistro di q. Per quanto riguarda la complessità computazionale bisogna tener conto che viene eseguita una ricerca di un nodo seguita da una ricerca di un minimo insieme con altre operazioni di costo costante. Ne segue che anche in questo caso il costo della funzione è lineare nell'altezza dell'albero in input.

8.3 Limitare l'altezza dell'albero

Come visto nella sezione precedente siamo in grado di eseguire le operazioni di ricerca, inserimento e cancellazione da un albero binario di ricerca in tempo lineare nell'altezza dell'albero. Tuttavia l'altezza h di un albero binario di ricerca di n nodi può essere anche lineare in n (basti pensare ad un albero in cui ogni nodo ha al più un figlio). All'opposto abbiamo il caso in cui tutti i nodi interni dell'albero hanno due figli in questo caso $n = \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ (si veda la dimostrazione della Proposizione 7.1 a pagina 68) ovvero $h = \log(n+1) - 1$. Se riuscissimo a garantire la proprietà che $h \in \Theta(\log n)$ ci assicureremmo il costo logaritmico su tutte le operazioni della struttura. In questa sezione illustreremo come mettere in pratica questa idea.

Un albero binario si dice *bilanciato* se la sua altezza è logaritmica nel numero dei nodi. Ovvero se l'altezza dell'albero h è $O(\log n)$. Quindi, per quanto detto sopra, una albero binario bilanciato ha altezza minima a parità di nodi e a meno di costanti moltiplicative.

Un tipo particolare di albero binario di ricerca bilanciato è l'albero *AVL* acronimo di Adelson-Velskii e Landis che lo introdussero nel 1962 (si veda [2]). Un albero AVL è un albero binario di ricerca che soddisfa ulteriori proprietà che lo rendono bilanciato. Sia u un nodo di un albero binario di ricerca T e sia $h(u)$ l'altezza del sotto-albero di T con radice u. L'albero T è un albero AVL se per ogni nodo u vale

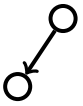
$$b(u) = |h(s(u)) - h(d(u))| \leq 1. \tag{8.1}$$

Ovvero le altezze dei due sotto-alberi sinistro e destro di u non possono differire per più di una unità. Questa proprietà garantisce che un albero AVL è bilanciato. Per questo motivo $b(u)$ viene detto *fattore di bilanciamento*.

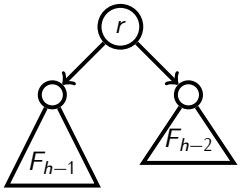
Teorema 8.3. *L'albero AVL è bilanciato.*

Dimostrazione. Bisogna dimostrare che l'altezza di un albero AVL di n nodi è $O(\log n)$. La dimostrazione procederà nel modo seguente: definiamo F_h come l'albero AVL minimo (nel numero di nodi) di altezza h e chiamiamo n_h il numero di nodi di F_h; dimostreremo che $n_h \approx 2^h$ quindi $n \geq 2^h$ ovvero $h \in O(\log n)$.

Per capire quanto vale n_h cerchiamo di capire com'è fatto F_h al variare di h. Se $h = 0$ allora F_h deve essere composto da un unico nodo. Se $h = 1$, F_h deve contenere almeno due nodi e l'albero in figura è un albero AVL di altezza 1 (omettiamo i pesi dei nodi in quanto siamo interessati solo alle altezze)



quindi $n_1 = 2$. In generale un albero AVL minimo di altezza $h \geq 2$ di radice r deve essere tale che uno dei due sotto-alberi sinistro o destro di r deve avere altezza $h-1$ e l'altro deve avere altezza $h-1$ o $h-2$ altrimenti non vale l'Equazione 8.1 su r. Escludiamo che quest'ultimo sotto-albero abbia altezza $h-1$ in quanto un albero AVL minimo di altezza inferiore ha meno nodi. Quindi senza perdita di generalità assumiamo che il sotto-albero sinistro di r ha altezza $h-1$ e quello destro altezza $h-2$. Questi, a loro volta, devono essere alberi AVL e affinché F_h sia minimo, devono essere di dimensione minima. Quindi la struttura di F_h è la seguente.



Ovvero i sotto-alberi sinistro e destro di r sono a loro volta alberi AVL minimi di altezza rispettivamente $h-1$ e $h-2$. Quindi

$$n_h = n_{h-1} + n_{h-2} + 1.$$

Esiste una relazione tra la successione n_h e la successione dei numeri di Fibonacci.[2] Dimostreremo che $n_h = f_{h+3} - 1$. La base dell'induzione segue dal fato che $n_0 = 1$ mentre $f_3 = 2$. Supponiamo che la tesi valga per tutti i valori di h inferiori ad h' allora

$$\begin{aligned} n_{h'} &= n_{h'-1} + n_{h'-2} + 1 \\ &= f_{h'+2} - 1 + f_{h'+1} - 1 + 1 \\ &= f_{h'+2} + f_{h'+1} - 1 \\ &= f_{h'+3} - 1. \end{aligned}$$

L'h-esimo numero di Fibonacci si può esprimere usando la formula di Binet (si veda [3])

$$f_h = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}$$

dove ϕ è una costante che vale circa 1.62. Da questo ricaviamo

$$\begin{aligned} n_h &= f_{h+3} - 1 = \frac{\phi^{h+3} - (1-\phi)^{h+3}}{\sqrt{5}} - 1 \\ &\geq \frac{\phi^{h+3} - 1}{\sqrt{5}} - 1 \end{aligned}$$

da cui segue, ignorando costanti moltiplicative, $h \leq \log n_h$, ovvero la tesi. □

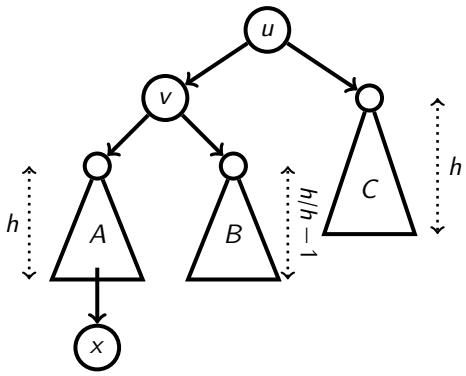
In generale un albero binario di ricerca non è un albero AVL in quanto una o più operazioni di inserimento e cancellazione potrebbero alterare il fattore di bilanciamento di qualche nodo interno e quindi, di fatto, sbilanciare l'albero. In particolare, l'inserimento o la cancellazione di un nodo da un albero AVL (quindi bilanciato) può sbilanciare tutti i sotto-alberi con radice i nodi nel percorso dalla radice fino al nodo inserito mentre tutti gli altri sotto-alberi restano bilanciati.

Si osservi che, poiché l'inserimento e la cancellazione alterano l'altezza dell'albero di al più una unità, inserire o cancellare un nodo da un albero AVL può introdurre un fattore di bilanciamento sui nodi di al più 2. Questo, come vedremo, ci consentirà di definire delle semplici operazioni di costo costante che ci permetteranno di portare il fattore di bilanciamento nella norma.

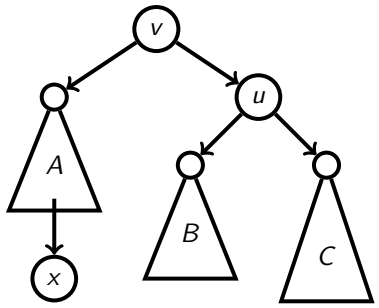
Per descrivere queste operazioni assumiamo che lo bilanciamento sia causato da una operazione di inserimento. Sia x il nuovo nodo ed u il primo nodo nel percorso da x alla radice che ha un fattore di bilanciamento maggiore di 1. Le azioni che verranno intraprese per ri-bilanciare l'albero dipenderanno dalla posizione in cui il nuovo nodo x è stato inserito. Infatti x può essere inserito nel sotto-albero sinistro o in quello destro di u. Tratteremo il caso in cui x viene inserito nel sotto-albero sinistro; l'altro caso, essendo simmetrico, può essere trattato in maniera analoga.

Sia v il figlio sinistro di u. Si hanno 3 sotto-casi diversi a secondo che x vada a finire nel sotto-albero sinistro o destro di v. Se x viene inserito nel sotto-albero sinistro di v avremo una situazione come quella mostrata in figura.

[2] La successione di Fibonacci f_h con $h = 0, 1, 2, \ldots$, è definita ricorsivamente nel seguente modo: $f_0 = 0$, $f_1 = 1$ e per $h \geq 2$, $f_h = f_{h-1} + f_{h-2}$.

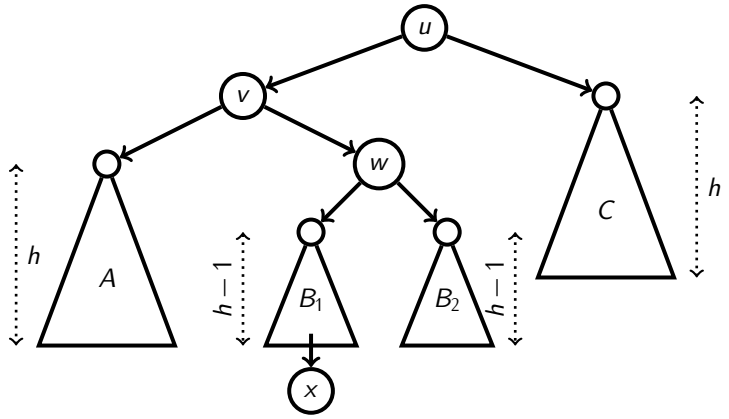


Se l'inserimento di x ha sbilanciato u allora precedentemente il fattore critico di u era 1, questo vuol dire che l'altezza del sotto-albero sinistro di u è $h+1$ e quello del sotto-albero destro è h (altrimenti l'albero non poteva essere AVL). Infine il sotto-albero sinistro di v deve avere altezza h ed il suo sotto-albero sinistro potrebbe avere altezza h oppure $h-1$. L'operazione che deve essere eseguita per ribilanciare l'albero è una *rotazione oraria* centrata sul nodo u, questa è illustrata dalla figura che segue.

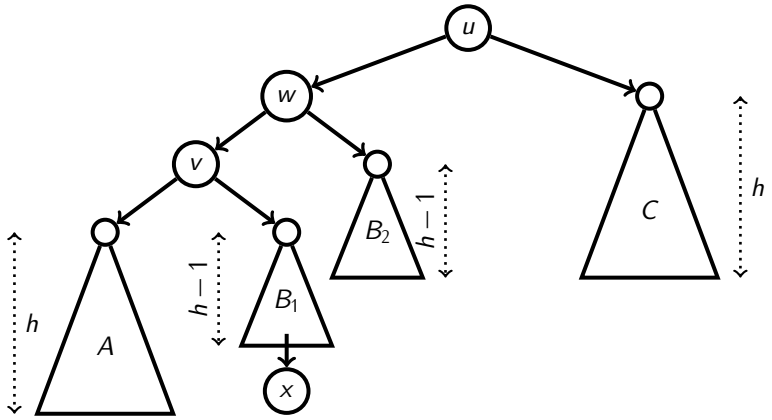


Ora si verifica che il sotto-albero con radice u ha altezza $h+1$ e fattore di bilanciamento 0 (nel caso in cui l'altezza di B sia h) oppure 1 (nel caso in cui l'altezza di B è $h-1$). L'albero con radice v ha altezza $h+2$ e, poiché il suo sotto-albero sinistro ha altezza $h+1$, esso ha fattore di bilanciamento 0. Inoltre dopo la rotazione la struttura continua ad essere un albero binario di ricerca. Infine l'operazione può essere eseguita in tempo costante in quanto devono essere modificati un numero costante di puntatori.

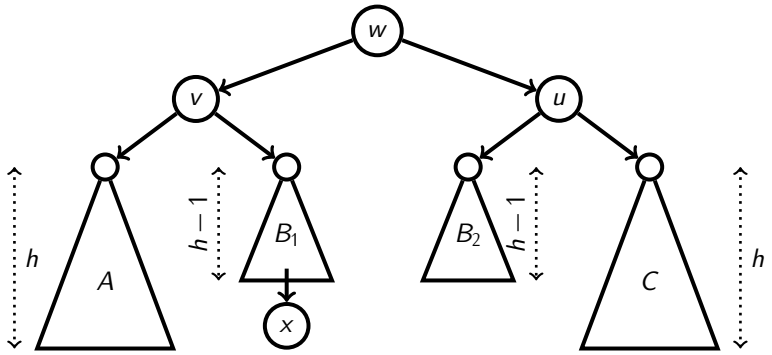
Passiamo al caso in cui in nuovo nodo x venga inserito nel sotto-albero destro di v che ha come radice w. In questo caso si devono distinguere due ulteriori sotto-casi: x viene inserito nel sotto-albero sinistro w oppure x viene inserito nel sotto-albero destro di w. Nella figura che segue è illustrato il primo caso.



Dal fatto che u è il primo nodo nel percorso radice-foglia ad essere sbilanciato dopo l'inserimento di x si deducono tutte le altezze dei sotto-alberi illustrati. Se C ha altezza h il sotto-albero con radice v deve avere altezza $h+2$ e quello con radice w deve avere altezza $h+1$. Poiché $b(v) \leq 1$, A non può avere altezza inferiore ad h; d'altra parte non può avere altezza superiore ad h altrimenti l'albero in u sarebbe stato sbilanciato anche prima dell'inserimento di x. Per lo stesso motivo gli alberi B_1 e B_2 devono avere altezza $h-1$. Una situazione di questo tipo si risolve ricorrendo a due operazioni di rotazione.



La prima è una *rotazione anti-oraria* centrata in v. Dopo questa prima rotazione il nodo u continua ad avere un fattore di bilanciamento 2. Ora però lo sbilanciamento è causato dal sotto albero sinistro del suo figlio sinistro. Questo sbilanciamento può essere risolto applicando una rotazione oraria centrata in u descritta nel caso precedente.



Ora i nodi risultano tutti bilanciati. Anche in questo caso, poiché ogni rotazione prevede un numero costante di operazioni, la complessità resta costante.

Si osservi che le due rotazioni appena descritte possono essere applicate anche nel caso in cui x venga inserito nel sotto-albero B_2.

Come già detto, nel caso in cui x venga inserito nel sotto-albero destro di u si avrebbero casi simmetrici rispetto a quelli appena descritti che si risolvono utilizzando le stesse operazioni di rotazione. In particolare, nel caso in cui x sia inserito nel sotto-albero destro del figlio destro v di u si applica una rotazione anti-oraria centrata in v; mentre se x viene inserito nel sotto-albero sinistro (con radice w) del figlio destro v di u si applicherà una rotazione oraria centrata in v seguita da una anti-oraria centrata in u.

Bibliografia

[1] Alessandro Bellini and Andrea Guidi. *Linguaggio C: Guida alla programmazione*. MCGraw-Hill, 2009.

[2] G.Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[3] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, 1989.

[4] Brian W. Kernighan and Dennis M. Ritchie. *Il linguaggio C. Principi di programmazione e manuale di riferimento*. Pearson Education Italia, 2004.