Permutazioni

1 Introduzione

Una *permutazione* su un insieme di n elementi (di solito $\{1, 2, \ldots, n\}$) è una funzione biiettiva dall'insieme in sé. In parole povere, è una regola che a ogni elemento dell'insieme, detto *lettera*, ne associa un altro (eventualmente l'elemento stesso) in maniera che a due lettere diverse non venga mai associata la stessa lettera. Le permutazioni vengono descritte in forma esplicita elencando su due righe le lettere (in ordine) e le lettere loro associate:

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix} \tag{1}$$

In questo caso, alla lettera 1 viene associato la lettera 3, alla lettera 2 la lettera 1 e così via. Notate che la lettera 4 viene associata a se stessa. Le condizioni che abbiamo dato fanno sì che la seconda riga debba contenere solo lettere distinte.

Un modo di pensare una permutazione è sotto forma di "spostamento": essa indica come una disposizione ordinata di oggetti deve essere spostata. Per esempio, se consideriamo la lista

$$c\ a\ b\ d$$

possiamo applicarle la permutazione (1): il carattere di posto 1 andrà al terzo posto e così via. La nuova lista risultante è

$$a\ b\ c\ d$$

La permutazione (1) è quindi quella che *ordina* la lista data. Torneremo più avanti su questo concetto.

Le permutazioni possono essere *composte*. Date due permutazioni sullo stesso numero di lettere possiamo cioè eseguirle in sequenza. Il risultato è ancora una permutazione. Esiste una permutazione che non sposta nulla:

$$\begin{pmatrix} 1 & 2 & 3 & \ldots & n \\ 1 & 2 & 3 & \ldots & n \end{pmatrix}$$

essa è detta la *permutazione identica*, e può essere effettuata prima o dopo qualunque altra permutazione senza alterare il risultato. Infine, data una permutazione ne esiste un'altra, l'*inversa*, che esegue lo scambio "opposto": per ottenere l'inversa di una permutazione basta scambiare la prima e la seconda riga e riordinare la prima. Per esempio, per ottenere l'inversa di (1) operiamo come segue:

$$\begin{pmatrix}1&2&3&4\\3&1&2&4\end{pmatrix} \rightsquigarrow \begin{pmatrix}3&1&2&4\\1&2&3&4\end{pmatrix} \rightsquigarrow \begin{pmatrix}1&2&3&4\\2&3&1&4\end{pmatrix} \tag{2}$$

È facile dimostrare che se componiamo (in qualunque ordine) una permutazione e la sua inversa otteniamo la permutazione identica.

2 Cicli

Una rappresentazione importante e molto compatta di una permutazione è in forma di *cicli*. Un *ciclo* è una lista di indici fra parentesi, e conveniamo che rappresenti la permutazione che associa a ogni indice nel ciclo quello successivo. Ad esempio, il ciclo

$$(1235)$$

rappresenta la permutazione che manda 1 in 2, 2 in 3 e così via fino a 5 in 1. Due cicli sono *digiunti* se non hanno lettere in comune. Per esempio, (123) e (45) sono disgiunti, ma (123) e (124) no. Per scrivere la composizione di permutazioni rappresentate da cicli, basta scrivere i cicli di seguito.

Non è difficile calcolare la permutazione risultante da una composizione di cicli: basta, per ogni lettera, "seguire il suo destino" lungo i vari cicli. Per esempio,

$$(123)(135)(24) = \begin{pmatrix}1&2&3&4&5\\4&5&3&2&1\end{pmatrix}. \tag{3}$$

Come abbiamo fatto il conto? Cominciamo da 1: il primo ciclo manda 1 in 2, il secondo non tocca il 2, il terzo manda 2 in 4: concludiamo che i tre cicli mandano 1 in 4. Il primo ciclo manda 2 in 3, il secondo 3 in 5, e il terzo non tocca 5: concludiamo che i tre cicli mandano 2 in cinque, e così via. Notate che alla fine del conto c'è un controllo di coerenza molto semplice: tutti i numeri nella seconda riga devono essere distinti.

Un teorema molto importante sulle permutazioni dice quanto segue.

Teorema 1 Ogni permutazione si può scrivere come una composizione di cicli disgiunti in modo unico a meno dell'ordine dei cicli stessi.

Come fare a ottenere una rappresentazione in cicli di una permutazione? Basta "seguire" una lettera qualunque fino a trovare un ciclo: per esempio, in (1) abbiamo che 1 va in 3, 3 va in 2 e 2 va in 1; quindi il primo ciclo che troviamo è (123). A questo punto non ci rimane che 4, che però va in sé, e formerebbe in ciclo di lunghezza 1. I cicli di lunghezza 1 per convenzione non si scrivono, e quindi la permutazione (1) si scrive (123).

Vediamo un esempio un po' più complicato:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 4 & 3 & 1 \end{pmatrix} \rightsquigarrow (125)(43)$$

Anche la permutazione (3) ha una rappresentazione in cicli digiunti: (1425). È un buon esercizio scrivere un programma che data una permutazione ne scrive la rappresentazione in cicli.

Dato un ciclo di lunghezza n, è possibile spezzarlo in cicli più corti, incrementando la lunghezza complessiva. Per esempio, $(123) = (12)(13)$. In generale, se avete un ciclo della forma $(i_1 i_2 \dots i_k)$ potete sempre scriverlo come $(i_1 i_2 \dots i_j)(i_1 i_{j+1} i_{j+2} \dots i_k)$ (con $j < k$). Questo processo può essere spinto fino a cicli di lunghezza due:

$$(i_1 i_2 \dots i_k) = (i_1 i_2)(i_1 i_3)(i_1 i_4) \dots (i_1 i_k).$$

3 Generazione di permutazioni casuali

La generazione di permutazioni casuali è importante per la generazione di input significativi. Inoltre, a volte permutare a caso i dati in input può servire per ridurre la probabilità che una cattiva configurazione dei dati faccia comportare male l'algoritmo (il caso tipico è la scelta dei pivot nel Quicksort). La rappresentazione più immediata per una permutazione è un vettore il cui i-esimo elemento contenga la lettera associata a i.

Per generare una permutazione su n elementi in tempo lineare si può usare il seguente codice C:

```
for(i=0; i<n; i++) p[i] = i;
for(i=n-1; i>0; i--)
   swapint(&p[i], &p[(int)((rand()/(1.0+RAND_MAX))*(i+1))]);
```

o il seguente codice Pascal:

```
FOR i:=1 TO n DO p[i] := i;
FOR i:=n DOWNTO 2 DO swapint(p[i], p[rand(i+1)]);
```

Abbiamo assunto che p sia un vettore di lunghezza n, e che la funzione `swapint` scambi il contenuto delle variabili che le sono passate.

Come funziona il codice? Descriviamo il comportamento del codice Pascal. Quello del codice C è simile, modulo il fatto che gli indici cominciano da zero e che dobbiamo fare un po' più di lavoro per ottenere un intero casuale ben distribuito.

In prima istanza, il vettore p contiene

1	2	3	4	...	n

A questo punto scandiamo il vettore da destra verso sinistra con l'indice i. Ogni volta cerchiamo un intero k compreso tra 1 e i (inclusi), e scambiamo il contenuto dell'k-esimo elemento del vettore con l'i-esimo. Qual è l'effetto?

In pratica stiamo costruendo la permutazione decidendo quale lettera associare a n, quindi quale lettera associare a $n-1$ e così via. A ogni passo, a sinistra dell'elemento di posizione i ci sono tutte le lettere che non abbiamo ancora scelto, e tra queste scegliamo chi associare alla lettera i. Al primo passo scegliamo uniformemente tra gli interi da 1 a n l'immagine di n. Quando scambiamo i due elementi, lasciamo nel posto n l'output corretto (l'immagine di n) e al tempo stesso lasciamo a sinistra di n le lettere che possono ancora essere scelti per essere associati a $n-1$, e così via.

Per esempio, fissiamo $n = 6$ e supponiamo di scambiare l'elemento di posto 6 con quello (scelto a caso) di posto 2. Il risultato sarebbe

1	6	3	4	5	2

A questo punto abbiamo scelto la lettera da associare a 6, e cioè 2. Al passo successivo, scegliamo a caso tra i cinque elementi più a sinistra la lettera da associare alla lettera 5, diciamo il primo:

5	6	3	4	1	2

Notate che i primi quattro elementi contengono (in modo disordinato, ma non importa perché tanto li scegliamo a caso) le lettere che possiamo associare a 4. Se, per esempio, scegliamo la seconda, arriviamo a

5	4	3	6	1	2

E così via. Alla fine dell'algoritmo abbiamo scelto una permutazione in modo uniforme e indipendente.

4 Come ricavare permutazioni da ordinamenti

A volte è utile ricavare da un vettore non ordinato `v` la permutazione che, applicata al vettore stesso, lo ordinerebbe. Per ottenere questo risultato basta applicare un qualunque algoritmo di ordinamento a un vettore fittizio `p` riempito inizialmente con gli indici 0, 1,. . . , $n - 1$ (o 1, 2, . . . , n se volete partire da 1). L'algoritmo di ordinamento viene a questo punto applicato al vettore `p`, ma i confronti tra l'elemento di posto i e quello di posto j non vengono effettuati confrontando `p[i]` e `p[j]`, ma piuttosto `v[p[i]]` e `v[p[j]]`. In questo modo, alla fine dell'algoritmo avremo un vettore che rappresenta la permutazione *inversa* di quella che ordinerebbe `v`: infatti, `p[i]` conterrà l'indice dell'elemento che finirebbe al posto i dopo l'ordinamento. Per ottenere la permutazione inversa di una data basta un ciclo—in C,

```
for(i=0; i<n; i++) q[p[i]] = i;
```

e in Pascal

```
FOR i:=1 TO n DO q[p[i]] := i;
```

dove `p` e `q` sono vettori di interi di lunghezza n, e `q`, alla fine del ciclo, contiene l'inverso della permutazione rappresentata da `p`.