generare tutte le 2^n sequenze binarie di lunghezza n, che possiamo equivalentemente interpretare come tutti i possibili sottoinsiemi di un insieme di n elementi.

Per illustrare questa corrispondenza, numeriamo gli elementi da 0 a $n-1$ e associamo il bit in posizione b della sequenza binaria all'elemento b dell'insieme fornito (dove $0 \leqslant b \leqslant n-1$): se tale bit è pari a 1, l'elemento b è nel sottoinsieme così rappresentato; altrimenti, il bit è pari a 0 e l'elemento non appartiene a tale sottoinsieme.

Durante la generazione delle 2^n sequenze binarie, memorizziamo ciascuna sequenza binaria A e utilizziamo la procedura `Elabora` per stampare A o per elaborare il corrispondente sottoinsieme. Notiamo che A viene riutilizzata ogni volta sovrascrivendone il contenuto ricorsivamente: il bit in posizione b, indicato con $A[b-1]$, deve valere prima 0 e, dopo aver generato tutte le sequenze con tale bit, deve valere 1, ripetendo la generazione.

Il seguente codice ricorsivo permette di ottenere tutte le 2^n sequenze binarie di lunghezza n: inizialmente, dobbiamo invocare la funzione `GeneraBinarie` con input $b = n$.

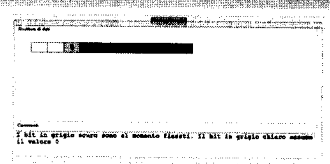
```
GeneraBinarie( A, b ):              ⟨pre: i primi b bit in A sono da generare⟩
  IF (b == 0) {
    Elabora( A );
  } ELSE {
    A[b-1] = 0;
    GeneraBinarie( A, b-1 );
    A[b-1] = 1;
    GeneraBinarie( A, b-1 );
  }
```

altri parametri possibili

000
010
100
110
001
011
101
111

ALVIE: generazione ricorsiva delle sequenze binarie

Osserva, sperimenta e verifica
`BinaryStringGeneration`



Il secondo esempio riguarda la generazione delle permutazioni degli n elementi contenuti in una sequenza A. Ciascuno degli n elementi occupa, a turno, l'ultima posizione in A e i rimanenti $n-1$ elementi sono ricorsivamente permutati. Per esempio, volendo generare tutte le permutazioni di $n = 4$ elementi a, b, c, d in modo sistematico, possiamo generare prima quelle aventi d in ultima posizione (elencate nella prima colonna), poi quelle aventi c in ultima posizione (elencate nella seconda colonna) e così via:

input	2° scambio	3° scambio	4° scambio
a b c d	a b d c	a d c b	d b c a
b a c d	b a d c	d a c b	b d c a
a c b d	a d b c	a c d b	d c b a
c a b d	d a b c	c a d b	c d b a
c b a d	d b a c	c d a b	c b d a
b c a d	b d a c	d c a b	b c d a

Restringendoci alle permutazioni aventi d in ultima posizione (prima colonna), possiamo permutare i rimanenti elementi a, b, c in modo analogo usando la ricorsione su questi tre elementi. A tal fine, notiamo che le permutazioni generate per i primi $n - 1 = 3$ elementi, sono identiche a quelle delle altre tre colonne mostrate sopra. Per esempio, se ridenominiamo l'elemento c (nella prima colonna) con d (nella seconda colonna), otteniamo le *medesime* permutazioni di $n - 1 = 3$ elementi; analogamente, possiamo ridenominare gli elementi b e d (nella seconda colonna) con d e c (nella terza colonna), rispettivamente. In generale, le permutazioni di $n - 1$ elementi nelle colonne sopra possono essere messe in corrispondenza biunivoca e, pertanto, ciò che conta sono il numero di elementi da permutare come riportato nel codice seguente. Invocando tale codice con parametro d'ingresso $p = n$, possiamo ottenere tutte le $n!$ permutazioni degli elementi in A:

```
GeneraPermutazioni( A, p ):   ⟨pre: i primi p elementi di A sono da permutare⟩
  IF (p == 0) {
    Elabora( A );   ⟵ altri parametri possibili.
  } ELSE {
    FOR (i = p-1; i >= 0; i = i-1) {
      Scambia( i, p-1 );
      GeneraPermutazioni( A, p-1 );
      Scambia( i, p-1 );
    }
  }
```

A = {1, 2, ..., n}
ciclo Hamilt.
TSP

Notiamo l'utilizzo di una procedura Scambia prima e dopo la ricorsione così da mantenere l'invariante che gli elementi, dopo esser stati permutati, vengono riportati al loro ordine di partenza, come può essere verificato simulando l'algoritmo suddetto.

ALVIE: generazione ricorsiva delle permutazioni

Osserva, sperimenta e verifica
PermutationGeneration