Generazione e conteggio di strutture combinatorie

corso di laurea in **Matematica**

Informatica Generale, Lezione **12(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Detour: combinazioni

Vediamo **3** modi per generare le **combinazioni** di n oggetti presi k a k (aiutino sull'Homework, dove avete le *permutazioni*).

È un insieme abbastanza "famoso": sono ad esempio sono combinazioni quelle del Lotto (& friends).

1. Generazione **ricorsiva**.
2. Generazione **ordinata** (con una funzione nextCbn che restituisce la prossima combinazione nell'ordine lessicografico)
3. Uso di **codifiche**/**decodifiche**.

Innanzitutto codifichiamo gli n oggetti con i numeri da 1 a n.

Gener. delle Combinazioni: ricorsione

L'idea è partire dal programma che le conta, e cioè dal programma che calcola ricorsivamente i coefficienti binomiali.

Cosa significano i **casi base**?

- ***k*=0**: non ho più elementi da sistemare e la *k*-upla è pronta.
- ***k*=*n***: devo sistemare *n*=*k* elementi su *k* posizioni e quindi ho un solo modo per farlo.

Cosa significano i **casi ricorsivi**?

- conto le combinazioni che ottengo **inserendo l'oggetto corrente**, cioè quelle in cui sistemo gli *n*-1 oggetti rimanenti in *k*-1 posizioni,
- più quelle in cui **non inserisco l'oggetto corrente** e quindi sistemo gli *n*-1 oggetti rimanenti nelle *k* posizioni.

```
def cbin(n, k):
# REQ: 0 ≤ k ≤ n
# ENS: return (n k)
    if k==0 or n==k: return 1
    return cbin(n-1, k-1) + cbin(n-1, k)
```

Esempio

I **terni** di **5** oggetti (numerati da 1 a 5) sono i seguenti:

[1, 2, 3], **[1**, 2, 4], **[1**, 2, 5], **[1**, 3, 4], **[1**, 3, 5]
[1, 4, 5], [2, 3, 4], [2, 3, 5], [2, 4, 5], [3, 4, 5]

È essenziale individuare un ordine per scriverle (e i programmi!)

Le combinazioni **che cominciano per 1** (in **verde&verde**) sono **6**, che è proprio $\binom{4}{2}$ e infatti si ottengono sistemando i numeri da 2 a 5 (che sono 4) nelle 2 posizioni rimanenti.

Le combinazioni **che non cominciano per 1** (in **blu**) sono **4**, che è proprio $\binom{4}{3}$ e infatti si ottengono sistemando i numeri da 2 a 5 (che sono 4) nelle **3** posizioni rimaste libere.

Devo poi riempire le **due caselle rimanenti** di quelle **che cominciano per 1**: **ce ne sono 3 che cominciano per 2** (cioè $\binom{3}{1}$=3 combinazioni in cui sistemo i numeri rimasti (3, 4, 5) nell'unica posizione libera e **3 che non cominciano per 2**, cioè le $\binom{3}{2}$=3 in cui sistemo i 3 numeri rimasti nelle 2 caselle rimaste libere.

Gener. delle Combinazioni: ricorsione

Per trasformare la funzione **cbin** in un programma che genera le combinazioni occorre **informazione supplettiva** sui **parametri**:

1. qual è il **prossimo numero** da **sistemare** *q*;

2. un **vettore c** che memorizza la **combinazione in costruzione**.

3. **la prossima posizione libera** *i* nella combinazione in costruzione.

Il resto segue dal pensiero induttivo/ricorsivo 😉

```
def generaCbn(n, k, q, c[], i):
    if k==0 || n==k:
        # k valori e k posizioni
        if n==k: riempi(q, c, i, k)
        print(c)
    else:
        c[i] = q # cmb che cominciano per q
        generaCbn(n-1, k-1, c, q+1, i+1)
            # cmb che cominciano per q+1
            # posizione i verrà riscritta
        generaCbn(n-1, k, c, q+1, i)
```

```
def riempi(k, q, c[], i):
    for j=i to k:
        c[j], q = q, q+1
```

Generaz. delle combinazioni: ordine

La prima combinazione è [1,2, …, k-1, k]. Ma chi è la prossima?

Riguardiamo il nostro esempio e ricoloriamolo:

[1, 2, **3**], [1, 2, **4**], [1, **2**, 5], [1, 3, **4**], [1, **3**, 5]
[**1**, 4, 5], [2, 3, **4**], [2, **3**, 5], [**2**, 4, 5], [3, 4, 5]

I numeri rossi sono quelli che **vengono incrementati** di 1 nella **combinazione successiva**: da quel punto in poi però, **la nuova combinazione è fatta di numeri consecutivi**.

Cioè la **minima combinazione** sulle **posizioni rimanenti**.
Ma questo è il lavoro che fa **`riempi`**.

Occorre risolvere il problema di **quale sia il prossimo numero da incrementare** e come accorgersi di aver finito.

Partendo da destra, si cerca il **primo indice** incrementabile e cioè:

- per l'ultima posizione, $c[k] < n$ permette di incrementare;
- per le posizioni "interne" dev'essere $c[k] + 1 < c[k+1]$

Se queste condizioni non sono mai verificate, ho finito.

Gener. delle Combinazioni: ordine

Nello pseudocodice, seguiremo un'idea ancora più semplice: nella parte "non incrementabile" è vero che $c[i] = n - k + i + 1$.

Se c'è un elemento incrementabile, avrò che $c[i] < n - k + i + 1$.

Se questa condizione non è mai verificata, sono sull'ultima combinazione.

Scritta la funzione **`prossimaCmb`**, è un gioco da ragazzi generarle tutte… 😉

```
def prossimaCbn(n, k, c[]):
    i = n-1
    while i > 0:
        if c[i] < n - k + i + 1:
            riempi(k, c[i]+1, c, i)
            return True
    return False
```

```
def generaCbn(n, k):
    c = alloca(c, k)
    riempi(k, 1, c, 0)
    repeat:
        print(c)
    until not prossimaCbn(n, k, c)
```

Generaz. delle combinazioni: codifica

In realtà questo metodo deriva **dagli stessi ragionamenti** del **metodo ricorsivo**: l'idea è codificare le combinazioni di cardinalità k di n numeri con i numeri naturali da 0 a $\binom{n}{k}$.

Se una combinazione **comincia per 1**, starà nelle **prime** $\binom{n-1}{k-1}$.

Se comincia per $q > 1$, devo **saltare** quelle che cominciano per **1, 2, …, q – 1** che saranno $\sum_{i=1,\ldots,q-1} \binom{n-q}{k-1}$.

E così via ripartendo dalla seconda posizione (riaggiornando k!).

Quest'idea permette **sia la codifica** che la **decodifica**.

Per implementarla **efficientemente**, conviene **pre-computarsi** una matrice dei coefficienti binomiali coinvolti (un **piccolo quadratino del triangolo di Tartaglia**) e "navigare" nella matrice.

Quest'idea è comunque **molto comoda**: dovendo scrivere programmi che trattano sistemi del Lotto, questi si possono **rappresentare come vettori di booleani indicizzati sulle codifiche** delle combinazioni…

Che fine hanno fatto le partizioni?

Ricordate il problema delle **partizioni di *n***? Vediamo le partizioni di 6 opportunamente ordinate (lessicograficamente) e colorate.

[1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 2], [1, 1, 1, 3], [1, 1, 2, 2],
[1, 1, 4], [1, 2, 3], [1, 5], [2, 4], [3, 3], [6]

Le **verdi** sono 1 seguito dalle partizioni di 5. Ma le **blu** cosa sono?

Sono le partizioni di 6 che non usano 1! Da questo posso trarre un'**idea ricorsiva**.

- **Esercizio**: **generare le partizioni** in analogia con le combinazioni, partendo dal programma ricorsivo scritto sotto.
- **Sfida**: scrivere la funzione **`prossimaPart`** (per **generarle in ordine** come fatto per permutazioni e combinazioni)

```
def part(n):
    return partAux(n, 1)
```

```
def partAux(n, k):
# conta partizioni di n che usano k come minimo numero
    if n==k: return 1 # c'è 1 partizione di n con min. k
    if n <k: return 0 # non c'è nessuna nessuna partizione
    return partAux(n-k, k) + partAux(n, k+1)
```