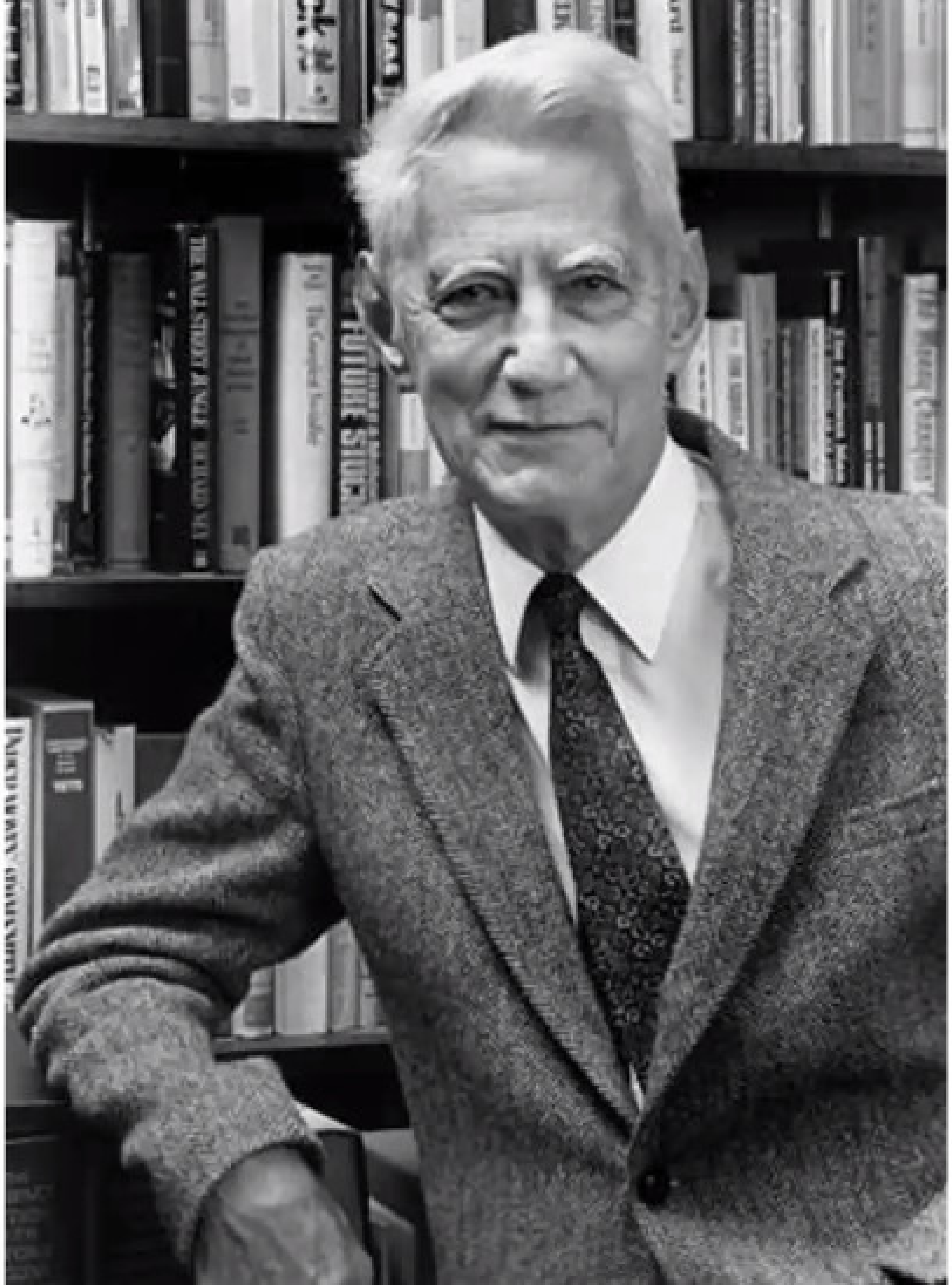
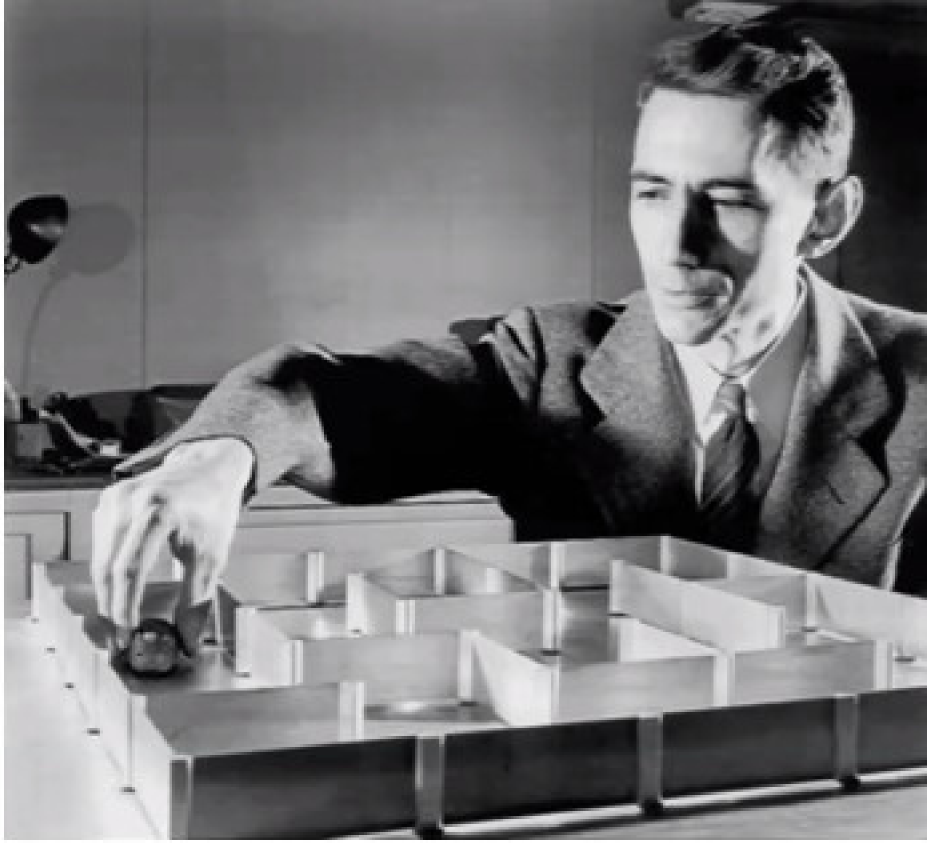
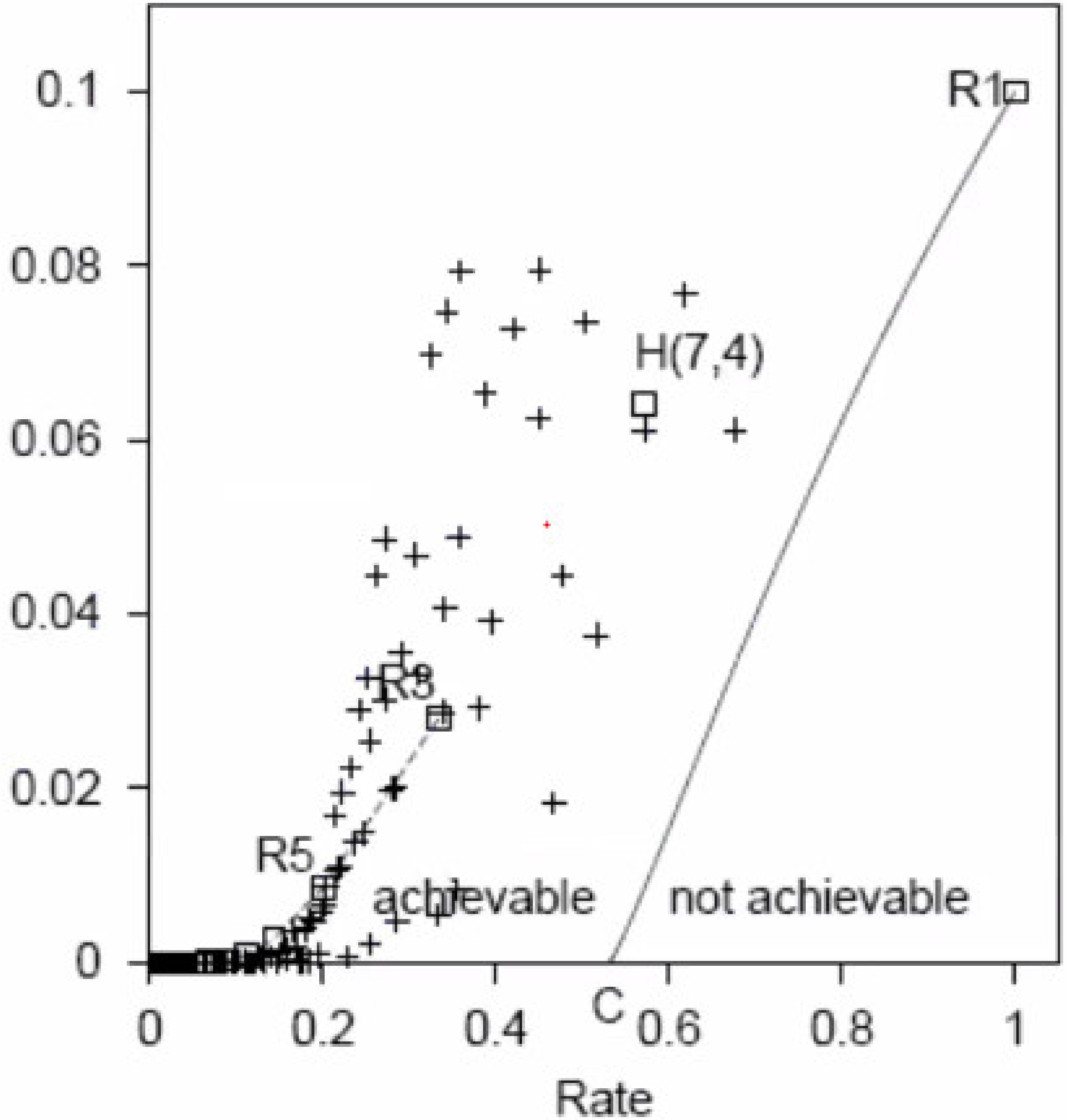


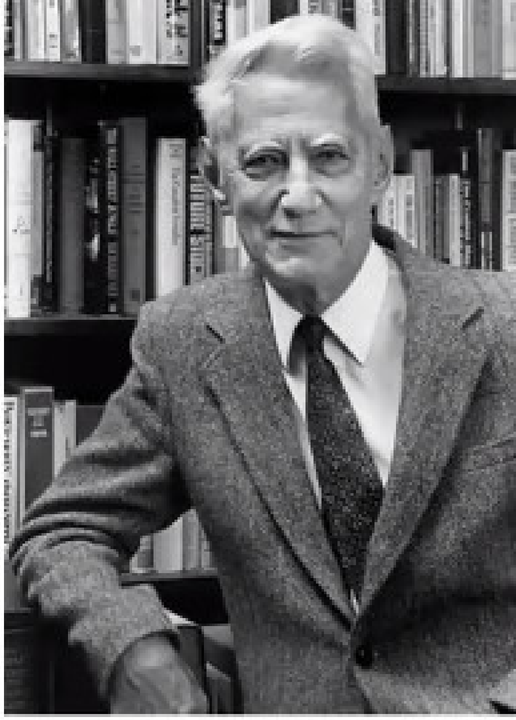
# Shannon, ancora lui!



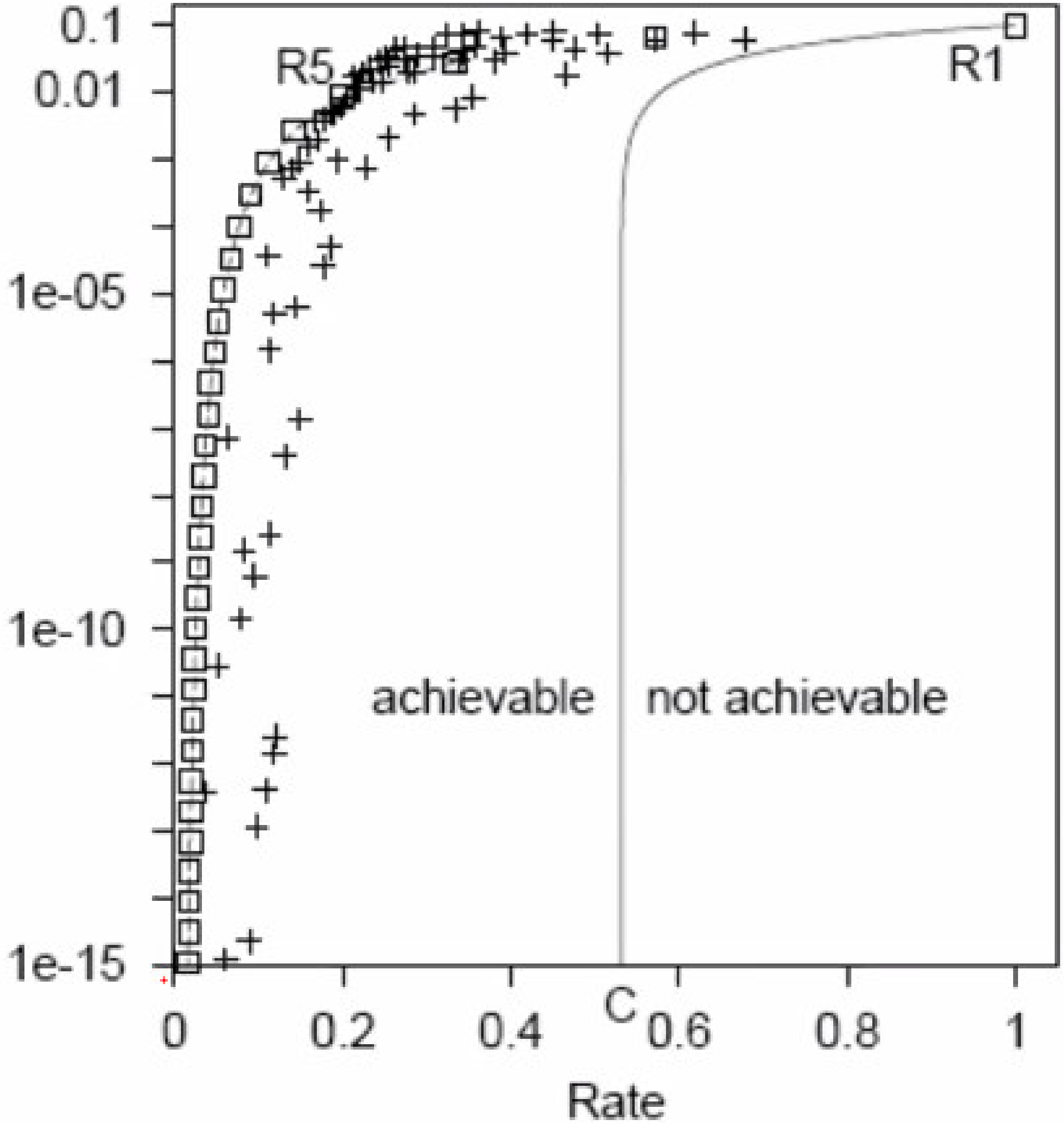
- Ha dimostrato che invece, anche se sembra impossibile, la curva dei codici migliori riesce ancora ad avere un ***data rate positivo***, anche se si vuole ***tasso di errore sempre più piccolo*** (!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!)

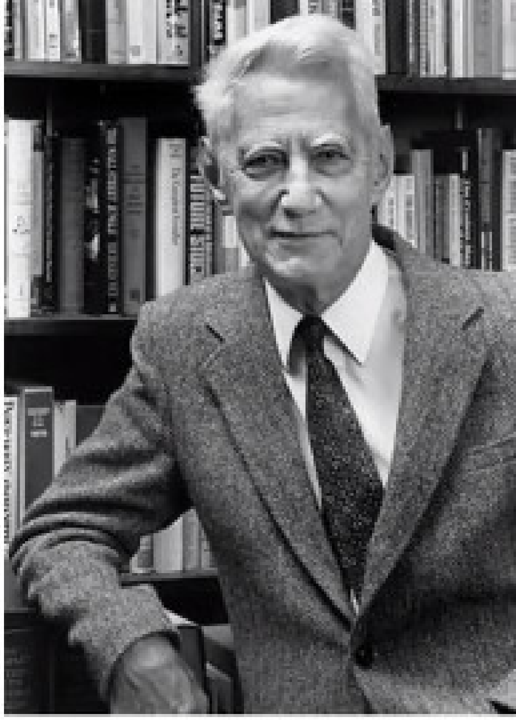
# Il limite di Shannon



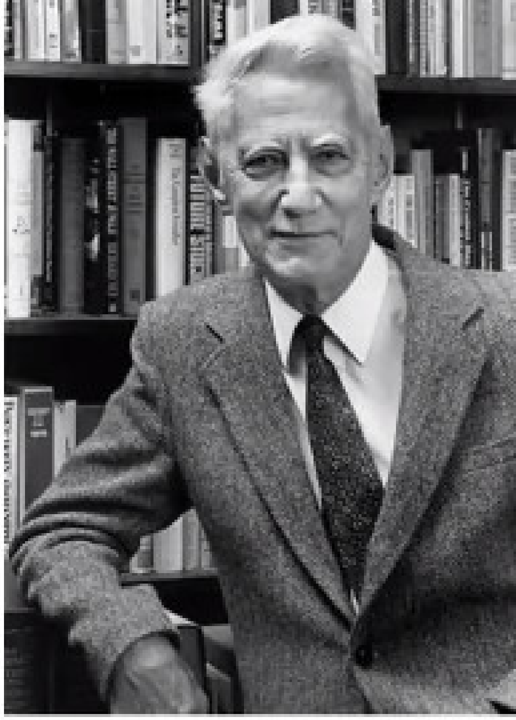



# Zona calda e limite (!)



# Il Teorema di Shannon



- Abbiamo visto un analogo nello strato fisico, qui è più potente:
- Dato un certo tasso d'errore **x**, vi dice che ci sono codici che possono arrivare ad un ***data rate massimo*** pari all'**entropia** del canale,

$$H_2(x) \equiv x \log \frac{1}{x} + (1-x) \log \frac{1}{(1-x)}$$

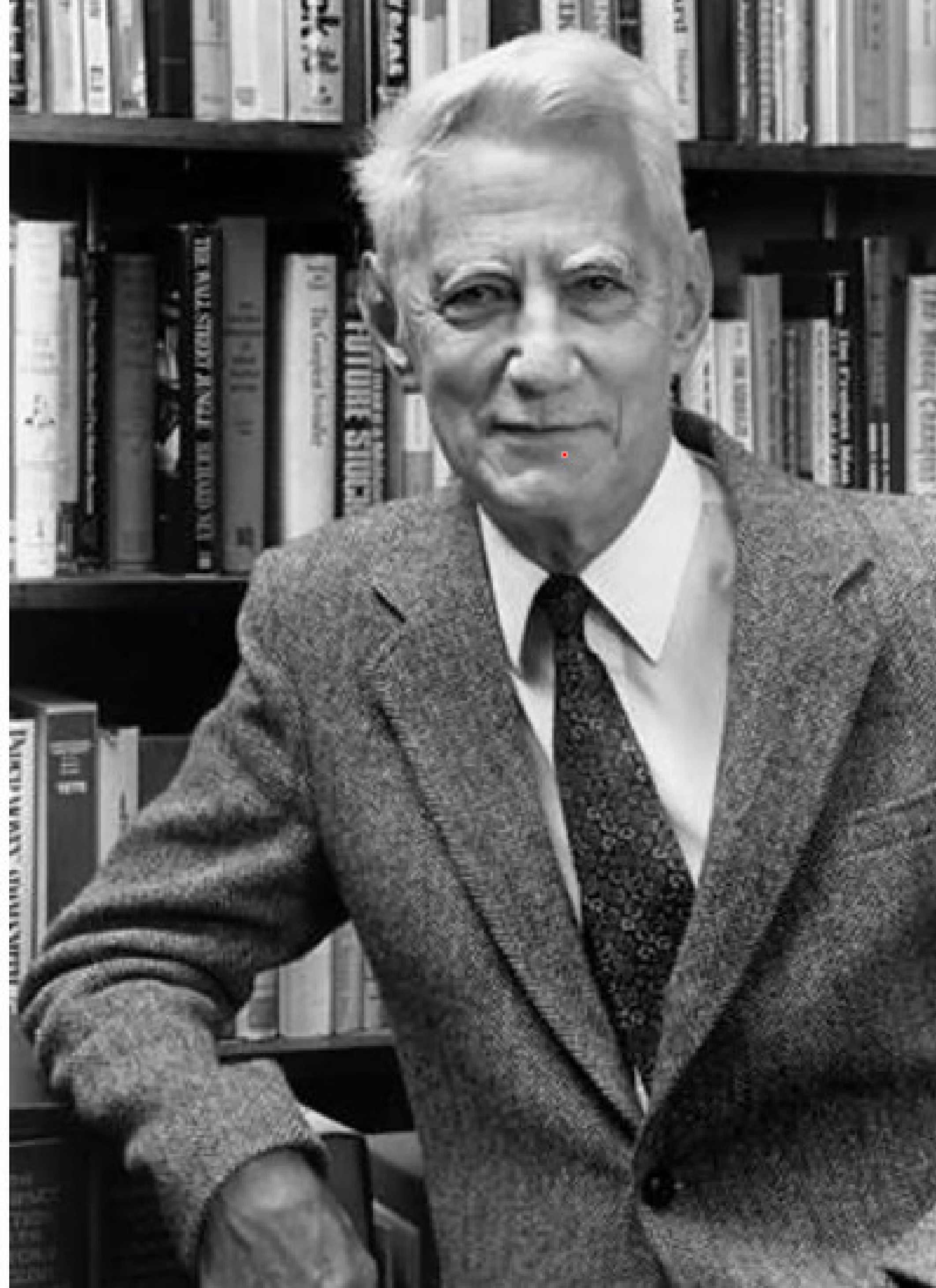
# Conseguenze

- Dischi rigidi "scassoni" del nostro computer con tasso d'errore **0.1** (!)
- Però vogliamo un tasso di errore decente, ad esempio ogni **10^-15** (ach!)
- → usando al meglio tecniche classiche tipo RAID (senza controllo dell'errore, quindi solo ridondanza), servono:

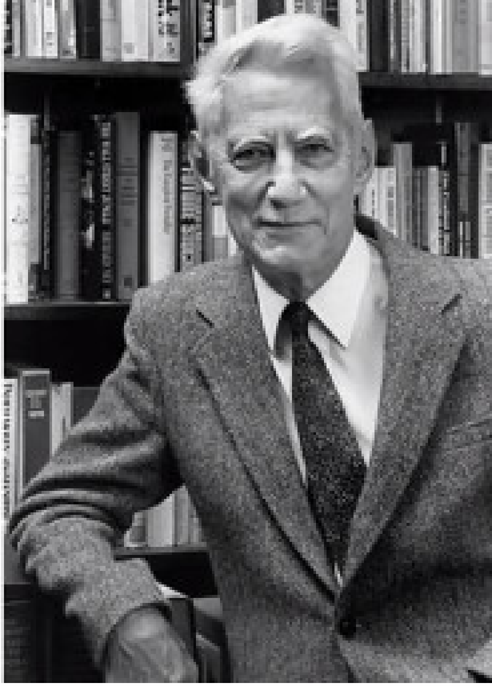
**60 DISCHI**

# Shannon?

- Guardate come sorride…



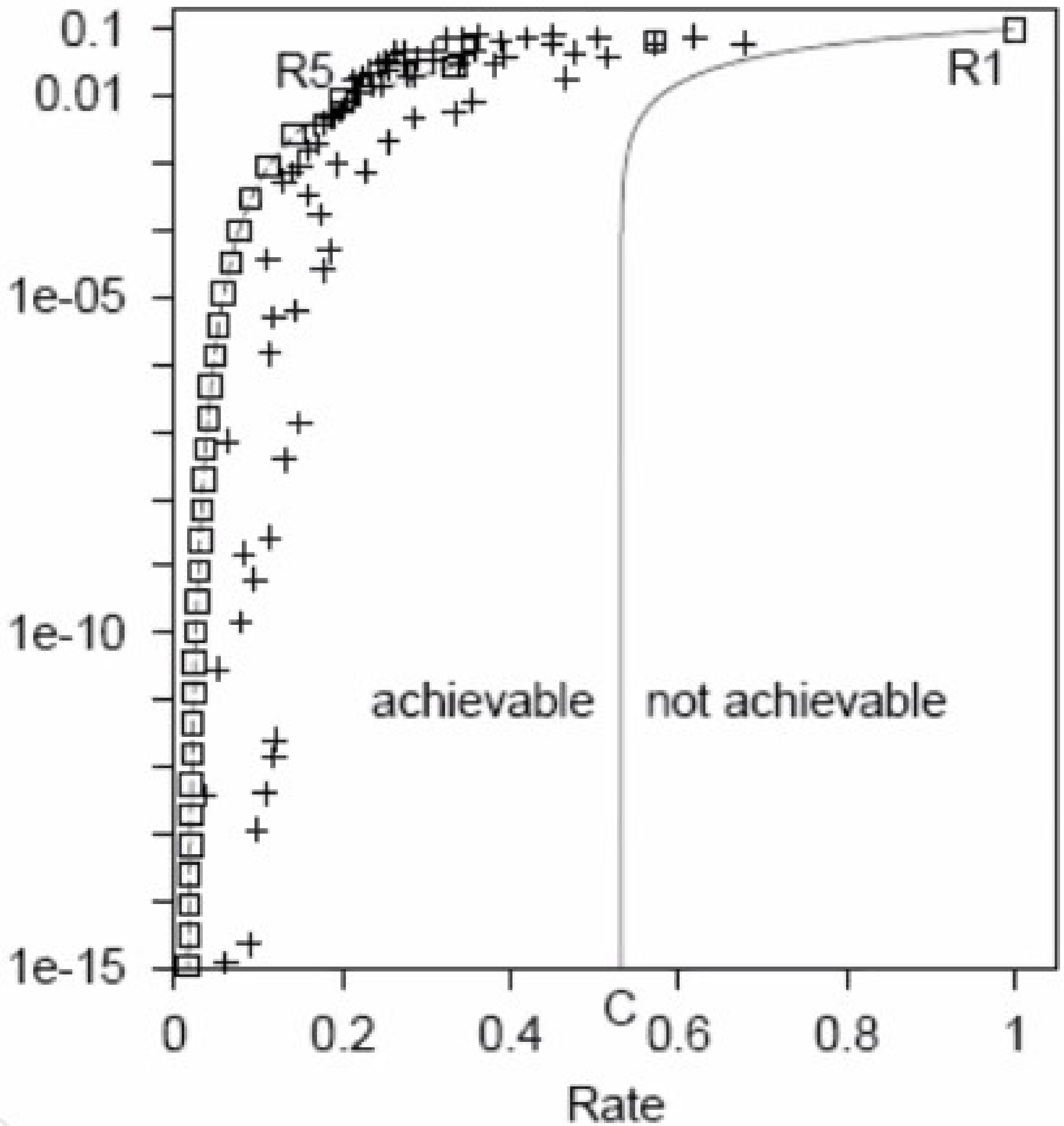
# Shannon!

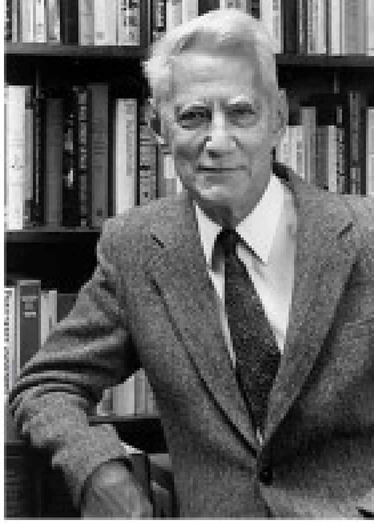


- Shannon ci dice che in realtà, c'è un codice per cui per passare dall'errore 0.1 a 10^-15 bastano meno di 60 dischi: ne bastano (usando la formula dell'entropia, H):

- **DUE (!!!!)**

# Abbiamo visto il limite di Shannon

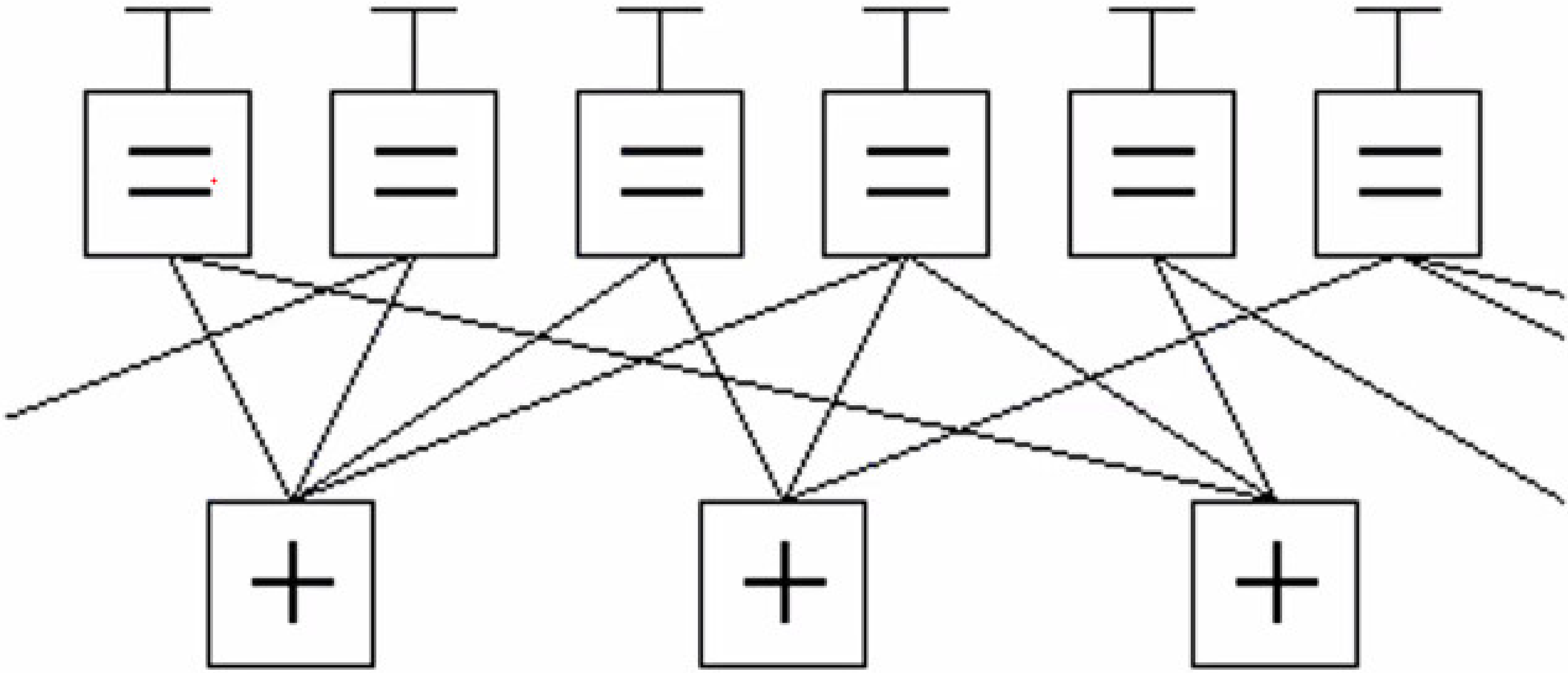




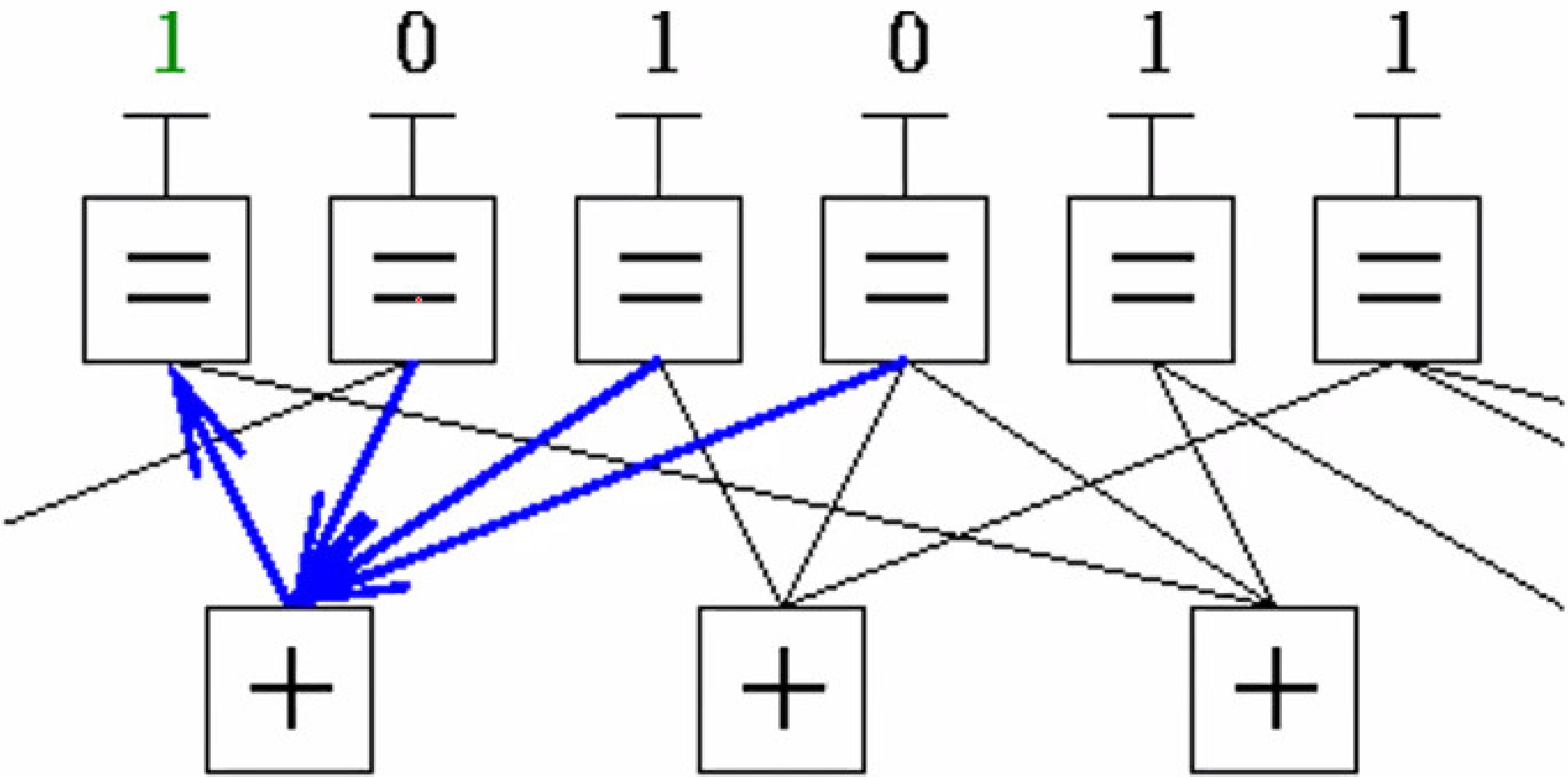
# Quali codici si avvicinano?

- Vediamo un esempio: i codici sparsi **LDPC**
- **LDPC** = **L**ow **D**ensity **P**arity **C**heck
- Uso: TV Digitale, Wi-Max

# LDPC



# LDPC (cont.)



# LDPC: check

$$\mathbf{z} = \mathbf{H}\mathbf{r} = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

# Potenza


- MOLTO vicina al limite di Shannon (!)
- DOVE SI PAGA…?
- Si paga nel seguente senso:
- Il ***decoding*** è **NP-COMPLETO** (!!!!!!)


# Vediamo ora…

- Un codice che ha un funzionamento simile a Reed-Solomon, solo che è molto più semplice perché fa solo error ***detection***, e non error ***correction***.
- In tal modo in ogni caso possiamo capire anche un pò il principio di funzionamento di Reed-Solomon

# CRC

- Sta per **C**yclic **R**edundancy **C**heck
- Basato sull'aritmetica polinomiale in base 2
- Tecnicamente, il GF(2)[] (l'anello dei polinomi sopra il campo finito con due elementi)

# Numeri come polinomi

- Possiamo vedere ogni numero binario come il ***corrispondente polinomio*** in GF(2)[], considerando ogni bit come il coefficiente di potenze sempre crescenti
- Esempi:

  **1011 → x^3 + x + 1**

  **11010 → x^4 + x^3 + x**

# Come funziona l'aritmetica in GF(2)[] ?

- Molto semplicemente!
- ***L'addizione*** è come fare lo ***xor !***
- La ***sottrazione*** è ***lo stesso dell'addizione*** (!!)

# Proviamo una divisione

- $$\frac{X^4 + X^2 + 1}{X + 1}$$
- $= X^3 + X^2$, con resto 1 (in GF(2))

Frame : 1 1 0 1 0 1 1 0 1 1

Generator: 1 0 0 1 1

Message after 4 zero bits are appended: 1 1 0 1 0 1 1 0 1 1 0 0 0 0

# Vediamo:

```
                    1 1 0 0 0 0 1 0 1 0
           _____________________________
1 0 0 1 1 | 1 1 0 1 0 1 1 0 1 1 0 0 0 0
            1 0 0 1 1
            ---------
              1 0 0 1 1
              1 0 0 1 1
              ---------
                0 0 0 0 1
                0 0 0 0 0
                ---------
                  0 0 0 1 0
                  0 0 0 0 0
                  ---------
                    0 0 1 0 1
                    0 0 0 0 0
                    ---------
                      0 1 0 1 1
                      0 0 0 0 0
                      ---------
                        1 0 1 1 0
                        1 0 0 1 1
                        ---------
                          0 1 0 1 0
                          0 0 0 0 0
                          ---------
                            1 0 1 0 0
                            1 0 0 1 1
                            ---------
                              0 1 1 1 0
                              0 0 0 0 0
                              ---------
                                1 1 1 0   <- Remainder
```

Transmitted frame: 1 1 0 1 0 1 1 0 1 1 1 1 1 0

# Vediamo:

- **X + 1** | X^4 + 0 + X^2 + 0 + 1

  X^4 + X^3

  ----------------

  X^3 + X^2

  X^3 + X^2

  ---------------

  0 + 0 +1

- → proprio X^3+X^2 con resto 1.

# L’idea di CRC

- E’ simile a quella dell’algoritmo di Reed-Solomon, e di una varietà di algoritmi simili:
- Il controllo di parità è dato dal resto d una divisione, però ***usando l’aritmetica di*** **GF(2)[]**

# Come si fa?

- Si sceglie un polinomio, il cosiddetto ***polinomio generatore*** (diciamo, **G(x)** )
- Abbiamo un ***messaggio*** **M(x)**
- Potremmo dividere M(x) per G(x), calcolare il ***resto*** **R(x)**, e quindi fare l'encoding trasmettendo **M(x)** seguito da **R(x)**

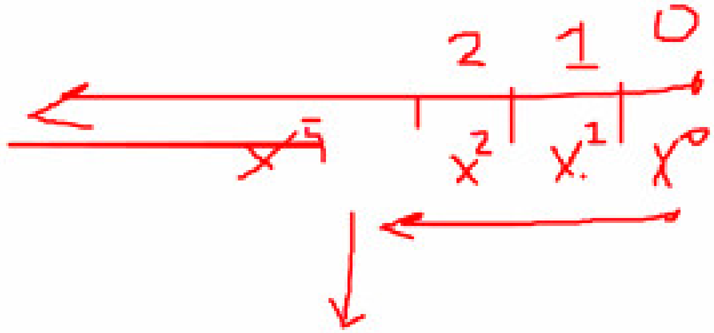
# Problemi?

- Il problema è ora scegliamo un numero G (polinomio G(x) ) qualunque...
- ... e quindi, se il messaggio M(x) è "***minore***" di G(x), allora il resto sarà M(x) stesso, quindi non abbiamo essenzialmente fatto nulla
- → resterebbero dei messaggi "scoperti", tutti quelli più piccoli di G(x)

# Come si fa (cont.)?

- Per risolvere questo problema, allora, moltiplichiamo il messaggio iniziale per x^(grado(G(x))
- Diciamo r=grado(G(x)) (G(x)=x^r + …. )
- Quindi, siamo sicuri che stavolta il numero ottenuto è “più grande” di G(x), e che il resto ora ha senso

# Nota




- Moltiplicare per x^r un polinomio equivale, ovviamente, a fare lo shift a sinistra di r posizioni.

# Quindi…

- L'*encoding* è semplice:
- Dividiamo x^r * M(x) per G(x)
- Calcoliamo il resto R(x)
- E poi trasmettiamo M(x) seguito da R(x)

Frame : 1 1 0 1 0 1 1 0 1 1

Generator: 1 0 0 1 1

Message after 4 zero bits are appended: 1 1 0 1 0 1 1 0 1 1 0 0 0 0

```
                1 1 0 0 0 0 1 0 1 0
         ----------------------------
   10011 | 1 1 0 1 0 1 1 0 1 1 0 0 0 0
           1 0 0 1 1
           ---------
             1 0 0 1 1
             1 0 0 1 1
             ---------
               0 0 0 0 1
               0 0 0 0 0
               ---------
                 0 0 0 1 0
                 0 0 0 0 0
                 ---------
                   0 0 1 0 1
                   0 0 0 0 0
                   ---------
                     0 1 0 1 1
                     0 0 0 0 0
                     ---------
                       1 0 1 1 0
                       1 0 0 1 1
                       ---------
                         0 1 0 1 0
                         0 0 0 0 0
                         ---------
                           1 0 1 0 0
                           1 0 0 1 1
                           ---------
                             0 1 1 1 0
                             0 0 0 0 0
                             ---------
                               1 1 1 0   <- Remainder
```

Transmitted frame: 1 1 0 1 0 1 1 0 1 1 1 1 1 0

# Vediamo: