BACKTRACKING e RICERCA ESAUSTIVA

- In molti casi non conosciamo algoritmi efficienti per risolvere il problema che abbiamo davanti.

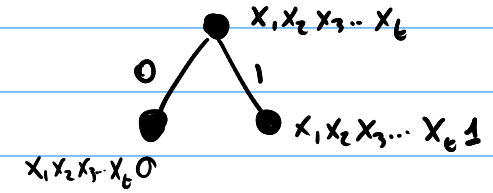
- In questi casi spesso l'unica opzione è la FORZA BRUTA (BRUTE FORCE) cioè l'esplorazione esaustiva di tutto lo SPAZIO DI RICERCA (SEARCH SPACE) o comunque di una grande fetta dello stesso.

- Visto che questo spazio può essere ENORME è necessario esplorarlo più velocemente possibile

ESERCIZIO 1 Data $n \geq 1$, stampare tutte le sequenze di n bit

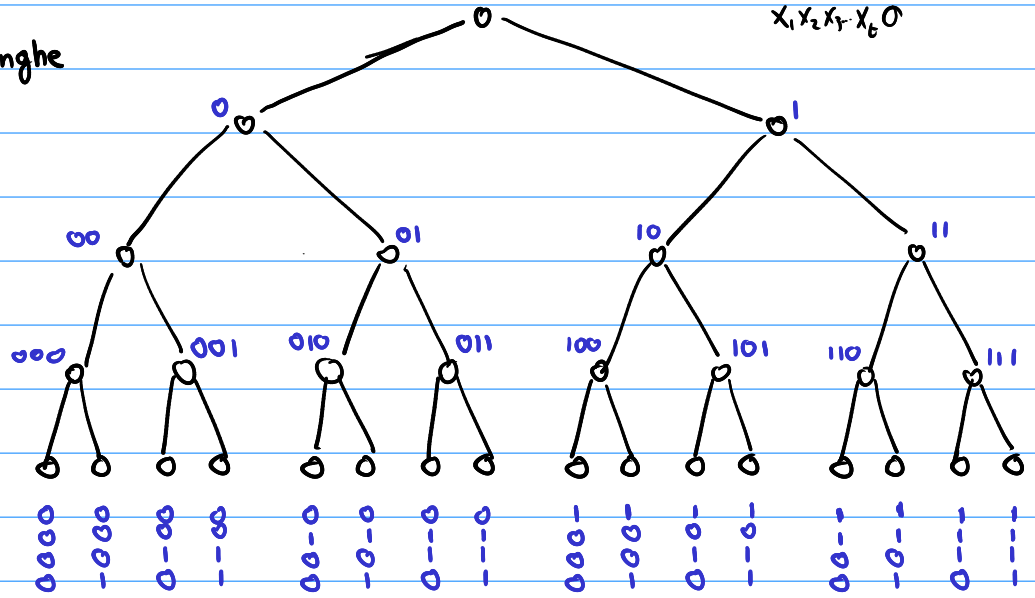
- Ci sono 2^n stringhe e ognuna richiede tempo $\Omega(n)$ per essere stampata. Quindi $\Omega(n \cdot 2^n)$ è il tempo necessario.

- L'idea è di usare un approccio ricorsivo



$n = 4$

$2^n = 16$ stringhe



```
7  def bits(n):
8      bits_rec(n,[])
9
10 def bits_rec(n,data):
11     if len(data)==n:
12         print(data)
13         return
14     data.append(0)
15     bits_rec(n,data)
16     data.pop()
17     data.append(1)
18     bits_rec(n,data)
19     data.pop()
```

ramo 0

ramo 1

Implementazione ricorsiva

Produce tutte le sequenze di lunghezza n con prefisso "data".

```
>>> bits(5)
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 1]
[0, 0, 0, 1, 0]
[0, 0, 0, 1, 1]
[0, 0, 1, 0, 0]
[0, 0, 1, 0, 1]
[0, 0, 1, 1, 0]
[0, 0, 1, 1, 1]
[0, 1, 0, 0, 0]
[0, 1, 0, 0, 1]
[0, 1, 0, 1, 0]
[0, 1, 0, 1, 1]
[0, 1, 1, 0, 0]
[0, 1, 1, 0, 1]
[0, 1, 1, 1, 0]
[0, 1, 1, 1, 1]
[1, 0, 0, 0, 0]
[1, 0, 0, 0, 1]
[1, 0, 0, 1, 0]
[1, 0, 0, 1, 1]
[1, 0, 1, 0, 0]
[1, 0, 1, 0, 1]
[1, 0, 1, 1, 0]
[1, 0, 1, 1, 1]
[1, 1, 0, 0, 0]
[1, 1, 0, 0, 1]
[1, 1, 0, 1, 0]
[1, 1, 0, 1, 1]
[1, 1, 1, 0, 0]
[1, 1, 1, 0, 1]
[1, 1, 1, 1, 0]
[1, 1, 1, 1, 1]
```

- Alle foglie dell'albero di ricorsione: costo $O(n)$
- Nei nodi interni: costo $O(1)$

foglie: 2^n # nodi interni: $2^n - 1$

- Costo totale $O(n 2^n)$ OTTIMO!

ESERCIZIO 2 Stampare tutte le stringhe in $\{0,1\}^n$ dove ci sono al massimo k uno.

con $0 \le k \le n$

E.s. $n=4$ $k=2$ 0000, 0001, 0010, 0011, 0100, 0101
0110, 1000, 1001, 1010, 1100

Strategia 1 Generare tutte le stringhe binarie come nell'esercizio precedente ma si stampano solo quelle stringhe che contengono al massimo k copie di "1".

Questo richiede circa $\Theta(n 2^n)$, che è troppo per k piccolo.

E.s. Stringhe di lunghezza n con $\le k$ copie di "1" sono

$$1 + n + \binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{k} \le n^k + 1$$

(0, 1, 2, 3, k copie di "1")

Def $$S(n,k) = \sum_{i=0}^{k} \binom{n}{i} = 1 + n + \cdots + \binom{n}{k}$$

La stampa di tutte le $S(n,k)$ stringhe richiede $\Omega(n \cdot S(n,k))$ tempo.

Q: È possibile scrivere un algoritmo che lo faccia in tempo $\Theta(n \cdot S(n,k))$?

BACKTRACKING: durante l'esplorazione dello spazio di ricerca delle stringhe binarie, possiamo provare a

TAGLIARE

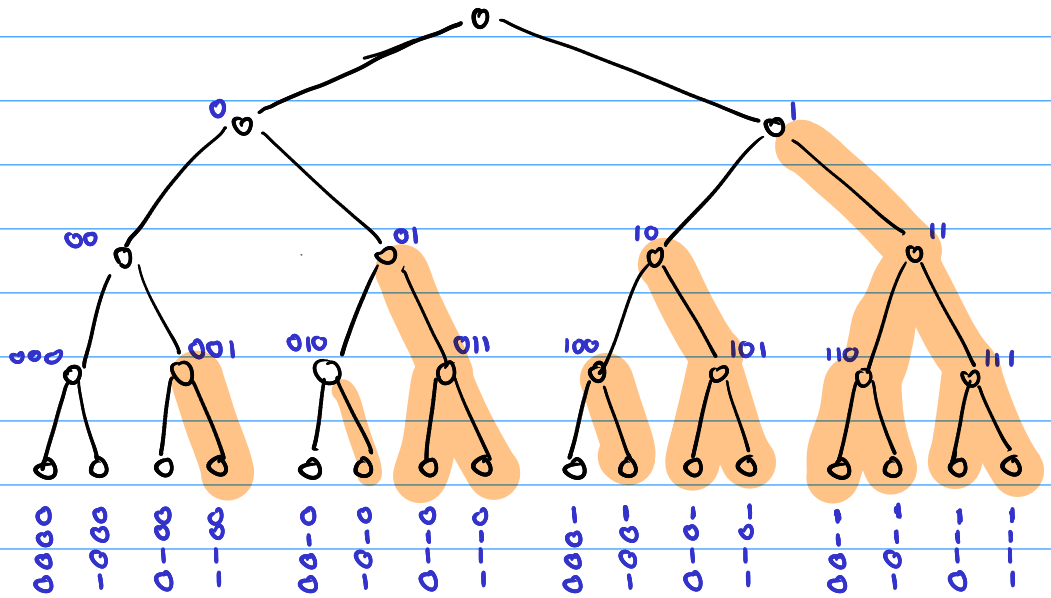
alcuni rami dell'albero che sappiamo già essere inutili,

TORNANDO INDIETRO SUI NOSTRI PASSI

per andare in una direzione alternativa

$n = 4$

$k = 1$



Dobbiamo evitare i rami che producono inevitabilmente stringhe con più di k copie di "1" e possiamo farlo IL PRIMA POSSIBILE, utilizzando una

FUNZIONE DI TAGLIO

- Durante la ricorsione teniamo traccia di quanti "1" sono ancora ammessi
- Quando non ne sono ammessi più, TAGLIAMO IL RAMO DESTRO

```
24 def bin_nk(n,k):
25     nk_rec(n,k,0,0,[])
26
27
28 def nk_rec(n,k,cnt0,cnt1,data):
29     if len(data)==n:
30         #print(data)
31         return
32     data.append(0)
33     nk_rec(n,k,cnt0+1,cnt1,data)
34     data.pop()
35     if cnt1 < k:
36         data.append(1)
37         nk_rec(n,k,cnt0,cnt1+1,data)
38         data.pop()
```

ramo 0

ramo 1

cnt0 e cnt1 mantengono il #0 e #1

- se #1 ≥ k allora non si aggiunge un altro "1"

Running time per $n = 30$ $k = 3$

- senza backtracking : 346.6 secondi [2.000.000.000 di nodi]
- con backtracking : 0,01 secondi [36.456 nodi]

OSSERVAZIONE Assumendo che valga, per la chiamata iniziale, $0 \le k \le n$ allora tutte le chiamate ricorsive producono ALMENO una stringa, perché si percorre sempre il ramo 0

Quindi tutte le foglie corrispondono ad una stringa dell'output

Verificate tutte e sole le stringhe di n bit con k bit uguali a "1" sono stampate dalla funzione.

PERCIÓ $\#\text{foglie} = S(n,k)$

$\#\text{nodi interni} \le n \cdot S(n,k)$ [perché l'altezza è $\le n$ dell'albero]

Operazioni: per foglia $O(n)$, per nodo interno $O(1)$

COMPLESSITÁ TOTALE: $O(n \cdot S(n,k))$ OTTIMO!

ESERCIZIO 3

Stampare tutte le stringhe binarie di lunghezza n che contengano esattamente k "1".

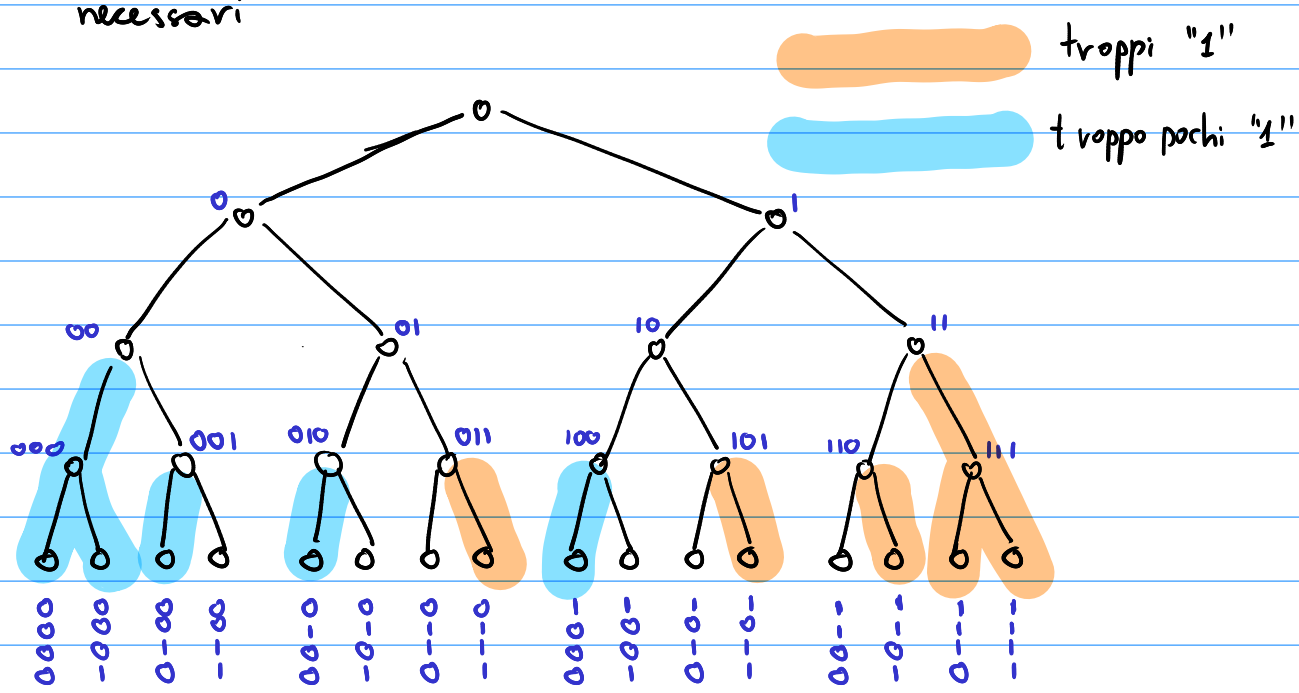
ESEMPIO $n=6$ $k=3$ sono 20 delle $2^6=64$ stringhe binarie

000111	001011	001101	001110
010011	010101	010110	011001
011010	011100	100011	100101
100110	101001	101010	101100
110001	110010	110100	111000

- In questo caso il nostro programma visto nell'esercizio 2 non taglia abbastanza rami, poiché genera stringhe con meno di k "1"
- Per TAGLIARE I RAMI che non possono portare a stringhe con esattamente k "1" posso eliminare quelli per cui
  - il numero di bit rimasti da decidere è minore degli "1" necessari

$n=4$

$k=2$



- $\#1 \geq k$ → taglio il ramo 1
- $\#0 \geq n-k$ → taglio il ramo 0

```python
def bin_nkexact(n,k):
    nkexact_rec(n,k,0,0,[])


def nkexact_rec(n,k,cnt0,cnt1,data):
    if len(data)==n:
        #print(data)
        return
    if cnt0 < n - k:
        data.append(0)
        nk_rec(n,k,cnt0+1,cnt1,data)
        data.pop()
    elif cnt1 < k:
        data.append(1)
        nk_rec(n,k,cnt0,cnt1+1,data)
        data.pop()
```

ramo 0

ramo 1

Prop $\#0 \geq n-k$ e $\#1 \geq k$ non possono avvenire simultaneamente in un nodo interno della ricorsione

dim $\#0 + \#1 \geq (n-k) + k = n$

Corollario $\#\text{foglie}$ = numero di stringhe di lunghezza n con esattamente k "1". $= \binom{n}{k}$

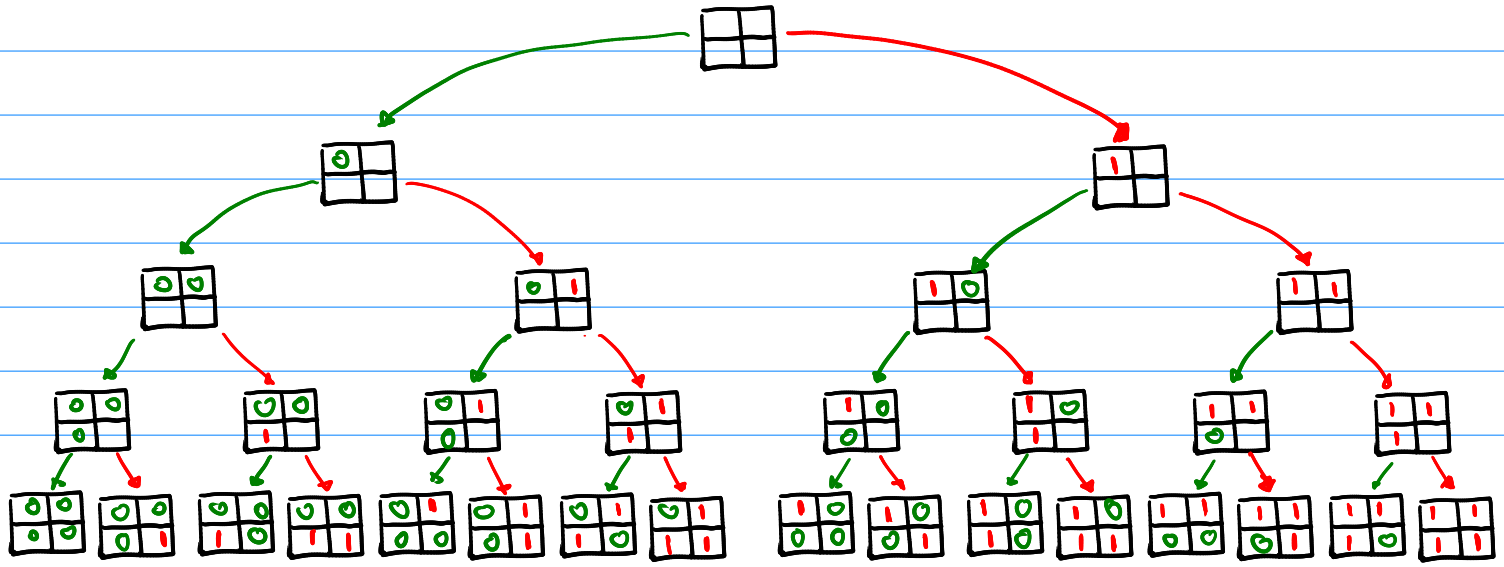
COMPLESSITÁ $O\left(n \cdot \binom{n}{k}\right)$ OTTIMA!

poiché $\#\text{nodi interni} \leq n \cdot \#\text{foglie}$

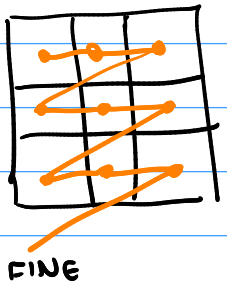
$O(1)$ operazioni per nodo interno

$O(n)$ operazioni per foglia

Esercizio 4 Generiamo tutte le matrici $n \times n$ con valori 0, 1



- Non è tanto differente dal generare stringhe di lunghezza n^2 tuttavia può far comodo mantenere la struttura di matrice
- Partiamo con la posizione $(0,0)$ e ad ogni passo nella ricorsione andiamo dalla posizione (i,j) alla posizione
  - $(i,j+1)$ se $j < n$
  - $(i+1,0)$ se $j = n-1$
- Siamo arrivati alla fine quando $i = n$




```
73 def bitmatrix(n):
74     M=creatematrix(n,n,fill=None)
75     bm_rec(n,0,0,M)
76
77 def bm_rec(n,i,j,data):
78     if i == n:
79         print(data)
80         return
81
82     if j+1 < n:
83         ni,nj = i,j+1
84     else:
85         ni,nj = i+1,0
86
87     data[i][j]=0
88     bm_rec(n,ni,nj,data)
89
90     data[i][j]=1
91     bm_rec(n,ni,nj,data)
92
93     data[i][j]=None
```

NODO FOGLIA

DETERMINA LA PROSSIMA COORDINATA DA RIEMPIRE

RAMO 0

RAMO 1

COMPLESSITÁ: ci sono 2^{n^2} matrici da stampare e ognuna richiede n^2 operazioni per essere stampata.
Quindi $\Omega(n^2 \cdot 2^{n^2})$ operazioni sono necessarie

L'algoritmo visto: $\Theta(1)$ operazioni nei nodi interni

$\Theta(n^2)$ operazioni sulle foglie

$\#\text{foglie} = 2^{n^2}$

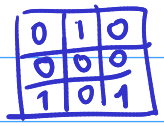
altezza della ricorsione $= n^2$

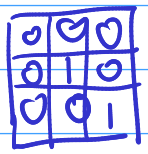
$\#\text{nodi interni} \leq \text{altezza} \times \#\text{foglie} \leq n^2 \cdot 2^n$

TOTALE: $\Theta(n^2 \cdot 2^{n^2})$

Esercizio 5 Progettare un algoritmo **OTTIMO** che data $n > 0$ stampi tutte le matrici binarie $n \times n$ che **NON CONTENGANO DUE 1 IN CELLE ADIACENTI** (dove intendiamo adiacenti in verticale, orizzontale e diagonale).

SI

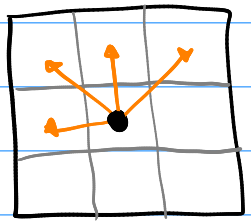




NO

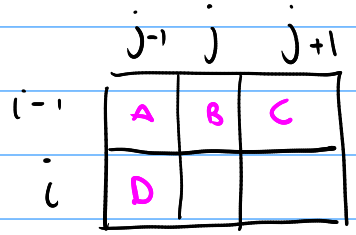
- La strategia esaustiva è quella di generare tutte le matrici binarie e poi verificare per ognuna se ci sono due 1 adiacenti.

- Quando bm_rec viene chiamata con argomenti (i,j) sappiamo che tutte le celle (i',j') con $i' < i$ e $i = i'$ e $j' < j$ sono state riempite e dobbiamo decidere il valore in posizione (i,j)

- 

  basta controllare quindi le celle nelle posizioni $(i-1,j-1)$ $(i-1,j)$ $(i-1,j+1)$ $(i,j-1)$ poiché le altre posizioni adiacenti non sono state ancora riempite e

  al momento di essere riempite controlleremo il contenuto di (i,j)




```python
def bitmatrix(n):
    M=creatematrix(n,n,fill=None)
    bm_rec(n,0,0,M)

def bm_rec(n,i,j,data):
    if i == n:
        print(data)
        return

    if j+1 < n:
        ni,nj = i,j+1
    else:
        ni,nj = i+1,0

    data[i][j]=0
    bm_rec(n,ni,nj,data)

    if bm_check(n,i,j,data):
        data[i][j]=1
        bm_rec(n,ni,nj,data)

    data[i][j]=None
```

```python
def bm_check(n,i,j,data):
    if i>0:
        if j>0 and data[i-1][j-1]==1:    # A
            return False
        if data[i-1][j]==1:              # B
            return False
        if j<n-1 and data[i-1][j+1]==1:  # C
            return False
    if j>0 and data[i][j-1]==1:          # D
        return False
    return True
```

COMPLESSITÁ: Sia $C(n)$ il numero di matrici $n \times n$ senza due "1" adiacenti

- Vedremo che l'algoritmo è ottimo anche se non sappiamo quanto vale $C(n)$
- Come negli esempi precedenti sappiamo che la FUNZIONE DI TAGLIO fa sì che OGNI nodo foglia costituisca una CONFIGURAZIONE VALIDA quindi:

$$C(n) = \#\text{foglie}$$

NON SPERATE CHE SIA SEMPRE COSÍ!!

- Dunque: il problema richiede tempo $\Omega(n^2 \cdot C(n))$
- $\#\text{foglie} = C(n)$ → costo $O(n^2)$ per foglia
- altezza albero $= n^2$
- $\#$ nodi interni $\leq n^2 \cdot \#$ foglie → costo $O(1)$ per ogni $\#$ nodo interno

Da qui costo totale

$$n^2 \cdot \#\text{Foglie} + 1 \cdot \#\text{nodi interni} \leq n^2 \cdot \#\text{foglie} + n^2 \cdot \#\text{foglie} = \boxed{O(n^2 \cdot C(n))}$$

- La funzione di taglio è essenziale per ridurre il tempo di calcolo perché elimina i RAMI INUTILI e quindi permette di tornare sui propri passi (lett. BACKTRACKING)
- Nella MIGLIORE DELLE IPOTESI si può evitare del tutto di costruire dei nodi che non portano a soluzioni.
- Questo non è sempre possibile e quindi il backtracking deve "ripensare" un maggior numero di scelte fatte.