Corso di laurea in Informatica
Progettazione d'algoritmi

Backtracking

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO 1

Progettare un algoritmo che prende come parametro un intero n e stampa tutte le stringhe binarie lunghe n.

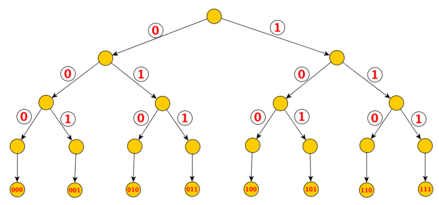
Ad esempio: per $n = 3$ l'algoritmo deve stampare $2^3 = 8$ stringhe:
000
001
010
100
011
101
110
111

OSSERVAZIONE:

- Le stringhe da stampare sono 2^n
- stampare una stringa richiede $\Theta(n)$

Il meglio che ci si può augurare per un algoritmo che risolve il problema è una complessità $\Omega(2^n \cdot n)$.

Albero di ricerca nel caso di $n = 3$



```python
def bk(n,i=0,sol=[]):
    if i==n:
        print(''.join(sol))
    else:
        sol.append('0')
        bk(n,i+1,sol)
        sol.pop()
        sol.append('1')
        bk(n,i+1,sol)
        sol.pop()

>>> bk(2)
00
01
10
11
```



```
def bk(n, i = 0, sol = [ ]) :
    if i == n :
        print(''.join(sol))
    else:
        sol.append('0')
        bk(n, i + 1, sol)
        sol.pop()
        sol.append('1')
        bk(n, i + 1, sol)
        sol.pop()
```

L'albero binario di ricorsione ha $2^n - 1$ nodi interni e 2^n foglie.

- ciascun nodo interno richiede tempo $O(1)$
- ciascuna foglia richiede tempo $O(n)$

La complessità dell'algoritmo è $O(2^n \cdot n)$

ESERCIZIO 2

Progettare un algoritmo che prende come parametro due interi n e k, con $0 \leq k \leq n$, e stampa tutte le stringhe binarie lunghe n che contengono al più k uni.

Ad esempio: per $n = 4$ e $k = 2$, delle $2^4 = 16$ stringhe lunghe n bisogna stampare le seguenti 11:

0000	0001	0010
0100	1000	0011
0101	1001	0110
1010	1100	

Un possibile algoritmo esaustivo di complessità $\Theta(2^n \cdot n)$:

```
def bk1(n, k, i = 0, sol = [ ]) :
    if i == n :
        if sol.count('1') <= k :
            print(''.join(sol))
    else:
        sol.append('0')
        bk1(n, k, i + 1, sol)
        sol.pop()
        sol.append('1')
        bk1(n, k, i + 1, sol)
        sol.pop()
```

La complessità dell'algoritmo è $\Theta(2^n \cdot n)$

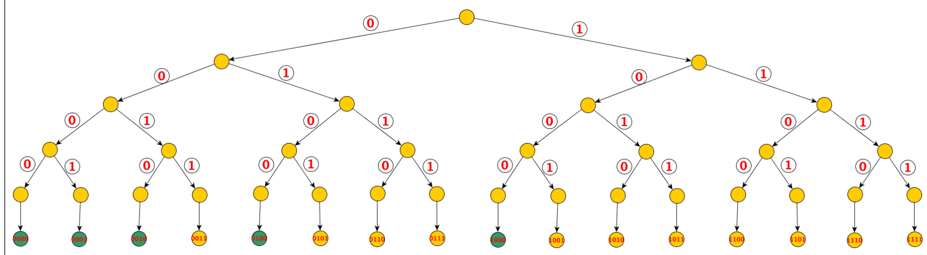
Indichiamo con $S(n,k)$ il numero di stringhe che bisogna stampare.

Un buon algoritmo per questo problema dovrebbe avere una complessità proporzionale alle stringhe da stampare, vale a dire $O(S(n,k) \cdot n)$ (poche stringhe = poco tempo).

Ad esempio per $k = 2$ si ha

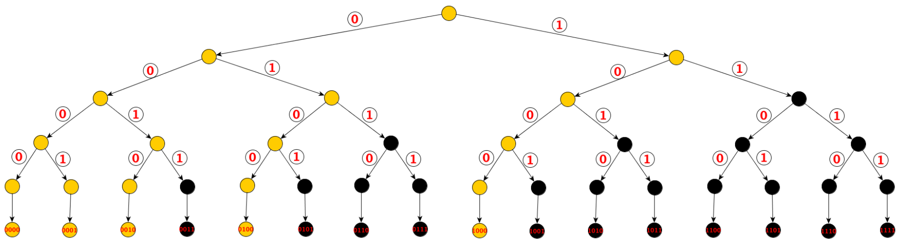
$$S(n,k) = 1 + n + \binom{n}{2} = \Theta(n^2)$$

e quindi un buon algoritmo per $k = 2$ dovrebbe avere complessità polinomiale $O(n^3)$ mentre l'algoritmo proposto ha complessità esponenziale $\Theta(2^n n)$ (indipendente da k)



Albero di ricorsione generato dall'algoritmo proposto nel caso di $n = 4$ e $k = 1$ con in verde le foglie che vanno stampate.

Osservazione:
Inutile generare nell'albero di ricorsione nodi che non hanno possibilità di portare a soluzioni da stampare.

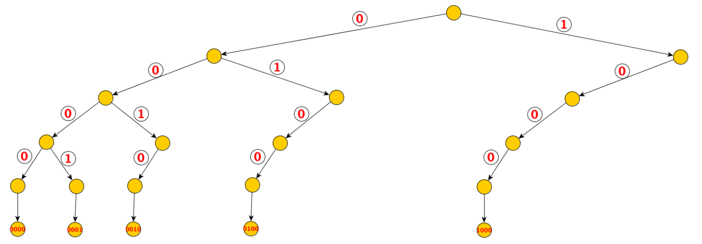


In nero i nodi che potrei evitare di generare nel caso in cui $n = 4$ e $k = 1$

algoritmo con un controllo sul numero di 1 presenti nel prefisso della stringa che si va generando:

```
def bk2(n, k, i = 0, tot1 = 0, sol = [ ]) :
    if i == n :
            print(''.join(sol))
    else:
            sol.append('0')
            bk2(n, k, i + 1, tot1, sol)
            sol.pop()
            if tot1 < k:
                    sol.append('1')
                    bk2(n, k, i + 1, tot1 + 1, sol)
                    sol.pop()
```

albero di ricorsione generato dalla funzione $bk2(n, k)$ con $n = 4$ e $k = 1$ grazie all'introduzione della funzione di taglio.



Per calcolare il tempo d'esecuzione del tuo programmino:

```
from time import time
start = time( )
# inserisci qui la funzione di cui vuoi testare il tempo di esecuzione
end = time( )
print(end − start)
```

Se invoco $bk1(24, 2)$ o $bk2(24, 2)$, in entrambi i casi verranno stampate le $1 + 24 + \frac{24 \cdot 23}{2} = 301$ stringhe binarie con al più 2 uni.

I tempi di calcolo per le due funzioni saranno invece molto diversi. In entrambi i programmi ho commentato il *print* che (come funzione di output è piuttosto costosa ma influisce sul tempo esattamente allo stesso modo per entrambi i programmi) e quello che ho ottenuto è quanto segue:

- **tempo di calcolo per $bk1(24, 2)$: 18.11323380470276**
- **tempo di calcolo per $bk2(24, 2)$: 0.002321004867553711**

Si consideri un algoritmo di enumerazione basato sul backtraking dove l'albero di ricorsione ha altezza h, il costo di una foglia è $g(n)$ e il costo di un nodo interno è $O(f(n))$.
Se l'algoritmo gode della seguente proprietà:

un nodo viene generato solo se ha la possibilità di portare ad una foglia da stampare.

Allora la complessità dell'algoritmo è proporzionale al numero di cose da stampare $S(n)$, più precisamente la complessità dell'algoritmo è:

$$O\left(S(n) \cdot h \cdot f(n) + S(n) \cdot g(n)\right)$$

questo perché:

- il costo totale dei nodi foglia sarà $O(S(n) \cdot g(n))$ (in quanto solo le foglie da enumerare verranno generate).
- i nodi interni dell'albero che verranno effettivamente generati saranno $O(S(n) \cdot h)$ (in quanto ogni nodo interno generato apparterrà ad un cammino che parte dalla radice e arriva ad una delle $S(n)$ foglie da enumerare).


```python
def bk2(n,k,i=0,tot=0,sol=[]):
    if i==n:
        print(''.join(sol))
    else:
        sol.append('0')
        bk2(n,k,i+1,tot,sol)
        sol.pop()
        if tot<k:
            sol.append('1')
            bk2(n,k,i+1,tot+1,sol)
            sol.pop()
```

Se l'algoritmo gode della seguente proprietà:

un nodo viene generato solo se ha la possibilità di portare ad una foglia da stampare.

allora la complessità dell'algoritmo è proporzionale al numero di cose da stampare $S(n)$, più precisamente la complessità dell'algoritmo è:

$$O\left(S(n) \cdot h \cdot f(n) + S(n) \cdot g(n)\right)$$

dove h è l'altezza dell'albero di ricorsione, $O(f(n))$ è il costo di un nodo interno e $g(n)$ è il costo di una foglia.

Per l'algoritmo codificato con la funzione $bk2(n, k)$. La proprietà di generare un nodo solo se questo può portare ad una delle $S(n, k)$ foglie da stampare è rispettata.
Inoltre $h = n$, $g(n) = \Theta(n)$, $f(n) = O(1)$.
Quindi la complessità dell'algoritmo è:

$$S(n, k) \cdot n \cdot O(1) + S(n, k) \cdot \Theta(n) = \Theta(S(n, k) \cdot n)$$

e l'algoritmo risulta ottimale.

La complessità dell'algoritmo è $O\left(n^{k+1}\right)$
infatti: $S(n, k) = \binom{n}{0} + \binom{n}{1} + \binom{n}{2} + \ldots + \binom{n}{k} < 2 \cdot n^k$

ESERCIZIO 3

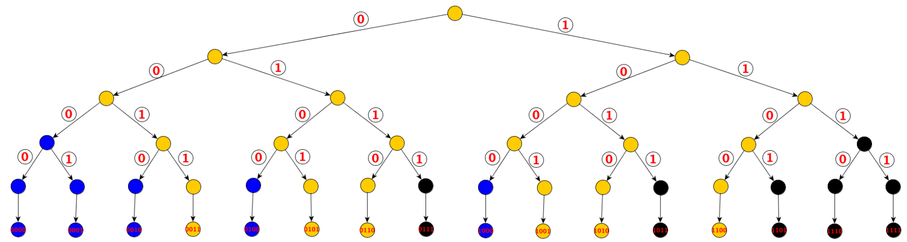
Progettare un algoritmo che prende come parametro due interi n e k, con $0 \leq k \leq n$, e stampa tutte le stringhe binarie lunghe n che contengono ESATTAMENTE k uni.

Ad esempio: per $n = 6$ e $k = 3$, delle $2^6 = 64$ stringhe lunghe n bisogna stampare le seguenti 20:

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

Rispetto all'esercizio precedente aggiungiamo ora una funzione di taglio anche quando al prefisso della soluzione viene aggiunto uno zero:
bisogna assicurarsi che sia sempre possibile completare quel prefisso in modo da ottenere una stringa da stampare.

```
def bk3(n, k, i = 0, tot1 = 0, sol = [ ]) :
    if i == n :
        print(''.join(sol))
    else:
        if tot1 + n - (i + 1) >= k :
            sol.append('0')
            bk3(n, k, i + 1, tot1, sol)
            sol.pop()
        if tot1 < k:
            sol.append('1')
            bk3(n, k, i + 1, tot1 + 1, sol)
            sol.pop()
```

l'albero di ricorsione generato da $bk3(n, k)$ con $n = 4$ e $k = 2$ con in nero i nodi che evito di generare grazie alla funzione di taglio sugli 1 ed in blu i nodi che evito di generare grazie alla funzione di taglio sugli 0.

Per l'algoritmo codificato con la funzione $bk3(n, k)$. La proprietà di generare un nodo solo se questo può portare ad una delle $S(n, k)$ foglie da stampare è rispettata.
Inoltre

- altezza dell'albero di ricorsione $h = n$,
- tempo richiesto da una foglia $g(n) = \Theta(n)$
- tempo richiesto da un nodo interno $f(n) = O(1)$

Quindi la complessità dell'algoritmo è:

$$S(n, k) \cdot n \cdot O(1) + S(n, k) \cdot \Theta(n) = \Theta(S(n, k) \cdot n)$$

e l'algoritmo risulta ottimale.

La complessità dell'algoritmo è $O\left(n^{k+1}\right)$
infatti: $S(n, k) = \binom{n}{k} < n^k$

ESERCIZIO 4

Progettare un algoritmo che prende come parametro un intero n e stampa tutte le matrici binarie n x n.

Ad esempio: per $n = 2$, bisogna stampare le seguenti $2^4 = 16$ matrici:

$\begin{bmatrix}0&0\\0&0\end{bmatrix}$ $\begin{bmatrix}0&0\\0&1\end{bmatrix}$ $\begin{bmatrix}0&0\\1&0\end{bmatrix}$ $\begin{bmatrix}0&0\\1&1\end{bmatrix}$ $\begin{bmatrix}0&1\\0&0\end{bmatrix}$ $\begin{bmatrix}0&1\\0&1\end{bmatrix}$ $\begin{bmatrix}0&1\\1&0\end{bmatrix}$ $\begin{bmatrix}0&1\\1&1\end{bmatrix}$

$\begin{bmatrix}1&0\\0&0\end{bmatrix}$ $\begin{bmatrix}1&0\\0&1\end{bmatrix}$ $\begin{bmatrix}1&0\\1&0\end{bmatrix}$ $\begin{bmatrix}1&0\\1&1\end{bmatrix}$ $\begin{bmatrix}1&1\\0&0\end{bmatrix}$ $\begin{bmatrix}1&1\\0&1\end{bmatrix}$ $\begin{bmatrix}1&1\\1&0\end{bmatrix}$ $\begin{bmatrix}1&1\\1&1\end{bmatrix}$

Algoritmo esaustivo1:

```
def bk(n, i = 0, j = 0, sol = []) :
  if i == n :
      for riga in range(n) : print(sol[riga])
      print( )
  else:
  # assegnamo sol[i][j] = 0
  if j == 0 :
      sol.append([0])
      bk(n, i, j + 1)
      sol.pop()
  else:
      sol[i].append(0)
      if j < n − 1 : bk(n, i, j + 1)
      else: bk(n, i + 1, 0)
      sol[i].pop()
  # assegnamo sol[i][j] = 1
  if j == 0 :
      sol.append([1])
      bk(n, i, j + 1)
      sol.pop()
  else:
      sol[i].append(1)
      if j < n − 1 : bk(n, i, j + 1)
      else: bk(n, i + 1, 0)
      sol[i].pop()
```


Algoritmo esaustivo2:

```
def mat1(n) :

  def bk(n, i = 0, j = 0) :
      if i == n :
          for riga in range(n) : print(sol[riga])
          print( )
      else:
          sol[i][j] = 0
          if j < n − 1 : bk(n, i, j + 1)
          else: bk(n, i + 1, 0)
          sol[i][j] = 1
          if j < n − 1 : bk(n, i, j + 1)
          else: bk(n, i + 1, 0)

  sol = [[0 for j in range(n)] for i in range(n)]
  bk(n)
```


Considerazioni sulla complessità

- L'algoritmo ha complessità $O\left(2^{n^2} n^2\right)$
  - l'albero di ricorsione è binario e di altezza n^2.
  - ha dunque $2^{n^2} - 1$ nodi interni e 2^{n^2} foglie
  - ciascun nodo interno richiede tempo $O(1)$ e ciascuna foglia richiede tempo $O\left(n^2\right)$
- Qualunque algoritmo per questo problema richiede tempo $\Omega\left(2^{n^2} n^2\right)$
  - le soluzioni da stampare sono 2^{n^2} e il tempo per stamare ciascuna di queste è $\Theta(n^2)$

L'algoritmo proposto è ottimo.

ESERCIZIO 5

Progettare un algoritmo che prende come parametro un intero n e stampa tutte le matrici binarie n x n in cui non compaiono uni adiacenti (in orizzontale, verticale o diagonale).

Ad esempio per $n = 3$ delle $2^9 = 512$ matrici quadrate 3×3 e binarie bisogna stampare le seguenti 34:

$$\begin{bmatrix}0&0&0\\0&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&0\\0&0&0\\0&0&1\end{bmatrix} \begin{bmatrix}0&0&0\\0&0&0\\0&1&0\end{bmatrix} \begin{bmatrix}0&0&0\\0&0&0\\1&0&0\end{bmatrix} \begin{bmatrix}0&0&0\\0&0&0\\1&0&1\end{bmatrix} \begin{bmatrix}0&0&0\\0&0&1\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&0\\0&0&1\\1&0&0\end{bmatrix}$$

$$\begin{bmatrix}0&0&0\\0&1&0\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&0\\1&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&0\\1&0&0\\0&0&1\end{bmatrix} \begin{bmatrix}0&0&0\\1&0&1\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&1\\0&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&1\\0&0&0\\0&0&1\end{bmatrix} \begin{bmatrix}0&0&1\\0&0&0\\0&1&0\end{bmatrix}$$

$$\begin{bmatrix}0&0&1\\0&0&0\\1&0&0\end{bmatrix} \begin{bmatrix}0&0&1\\0&0&0\\1&0&1\end{bmatrix} \begin{bmatrix}0&0&1\\1&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}0&0&1\\1&0&0\\0&0&1\end{bmatrix} \begin{bmatrix}0&1&0\\0&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}0&1&0\\0&0&0\\0&1&0\end{bmatrix} \begin{bmatrix}0&1&0\\0&0&0\\1&0&0\end{bmatrix}$$

$$\begin{bmatrix}0&1&0\\0&0&0\\1&0&1\end{bmatrix} \begin{bmatrix}1&0&0\\0&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}1&0&0\\0&0&0\\0&0&1\end{bmatrix} \begin{bmatrix}1&0&0\\0&0&0\\0&1&0\end{bmatrix} \begin{bmatrix}1&0&0\\0&0&0\\1&0&0\end{bmatrix} \begin{bmatrix}1&0&0\\0&0&0\\1&0&1\end{bmatrix} \begin{bmatrix}1&0&0\\0&0&1\\0&0&0\end{bmatrix}$$

$$\begin{bmatrix}1&0&0\\0&0&1\\1&0&0\end{bmatrix} \begin{bmatrix}1&0&1\\0&0&0\\0&0&0\end{bmatrix} \begin{bmatrix}1&0&1\\0&0&0\\0&0&1\end{bmatrix} \begin{bmatrix}1&0&1\\0&0&0\\0&1&0\end{bmatrix} \begin{bmatrix}1&0&1\\0&0&0\\1&0&0\end{bmatrix} \begin{bmatrix}1&0&1\\0&0&0\\1&0&1\end{bmatrix}$$

Nota che una matrice binaria sol deve essere stampata solo se e solo se per ogni cella $sol[i][j]$ che contiene un 1 tutte le celle adiacenti che la precedono nell'ordinamento per righe per colonne (vale a dire le potenziali 4 celle $sol[i-1][j-1]$, $sol[i-1][j]$, $sol[i-1][j+1]$ e $sol[i][j-1]$) contengono zeri.

0	0	0
0	1	_
_	_	_

$i-1, j-1$	$i-1, j$	$i-1, j+1$
$i, j-1$	i, j	_
_	_	_

Ecco di seguito una semplice funzione che prende come parametro la matrice $n \times n$ e in tempo $O(n^2)$ restituisce $True$ se e solo se da stampare.

```
def buona(sol) :
    n = len(sol)
    for i in range(n) :
        for j in range(n) :
            if sol[i][j] == 1 :
                if i > 0 and j > 0 and sol[i − 1][j − 1] == 1 : return False
                if i > 0 and sol[i − 1][j] == 1 : return False
                if i > 0 and j < n − 1 and sol[i − 1][j + 1] == 1 : return False
                if j > 0 and sol[i][j − 1] == 1 : return False
    return True
```


Algoritmo esaustivo:

```
def mat2(n) :

    def bk(n, i = 0, j = 0) :
        if i == n :
            if buona(sol):
                for riga in range(n) : print(sol[riga])
                print( )
        else:
            sol[i][j] = 0
            if j < n − 1 : bk(n, i, j + 1)
            else: bk(n, i + 1, 0)
            sol[i][j] = 1
            if j < n − 1 : bk(n, i, j + 1)
            else: bk(n, i + 1, 0)

    sol = [[0 for j in range(n)] for i in range(n)]
    bk(n)
```

L'algoritmo ha complessità $\Theta\left(2^{n^2} n^2\right)$ anche se le matrici da stampare sono molto meno di 2^{n^2}.

Anzicché generare la matrice *sol* e poi controllare se *sol* va stampata o meno procediamo generando *sol* in modo tale da assicurarci che il risultato sarà una matrice da stampare

Nella generica cella $sol[i][j]$ della matrice è possibile inserire sia 0 che un 1:

- inserire uno 0 non crea alcun problema, quindi in questo caso non abbiamo bisogno di ricorrere a funzioni di taglio.
- inserire un 1 è invece possibile solo se nessuna della potenziali celle $sol[i-1][j-1]$, sol[i-1][j], $sol[i-1][j+1]$ e $sol[i][j-1]$ contiene un 1. In questo caso quindi inseriamo una funzione di taglio che effettua il controllo.

Algoritmo di backtraking:

```
def mat3(n):

    def bk(n, i = 0, j = 0):
        if i == n:
            for riga in range(n): print(sol[riga])
            print()
        else:
            sol[i][j] = 0
            if j < n - 1: bk(n, i, j + 1)
            else: bk(n, i + 1, 0)
            if (i - 1 < 0 or j - 1 < 0 or sol[i - 1][j - 1] == 0) and \
              (i - 1 < 0 or sol[i - 1][j] == 0) and \
              (i - 1 < 0 or j + 1 > n - 1 or sol[i - 1][j + 1] == 0) and \
              (j - 1 < 0 or sol[i][j - 1] == 0):
                sol[i][j] = 1
                if j < n - 1: bk(n, i, j + 1)
                else: bk(n, i + 1, 0)

    sol = [[0 for j in range(n)] for i in range(n)]
    bk(n)
```


Considerazioni sulla complessità dell'algoritmo *mat*3

Siano $S(n)$ le matrici da stampare

- L'algoritmo ha complessità $O\left(S(n) \cdot n^2\right)$
  - l'albero di ricorsione è binario e di altezza n^2.
  - solo i nodi che portano ad una delle $S(n)$ soluzioni vengono effettivamente generati
  - i nodi interni all'albero di ricorsione effettivamente generati saranno $O(S(n) \cdot n^2)$ e le foglie effettivamente generate saranno $S(n)$
  - ciascun nodo interno richiede tempo $O(1)$ e ciascuna foglia richiede tempo $O\left(n^2\right)$
  - il tempo in totale sarà $O\left(S(n) \cdot n^2\right)+ O\left(S(n) \cdot n^2\right)=O\left(S(n) \cdot n^2\right)$
- Qualunque algoritmo per questo problema richiede tempo $\Omega\left(S(n) \cdot n^2\right)$
  - le soluzioni da stampare sono $S(n)$ e il tempo per stampare ciascuna di queste è $\Theta(n^2)$

L'algoritmo proposto è ottimo.