Corso di laurea in Informatica
Progettazione d'algoritmi

Tecnica Backtracking 2

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

ESERCIZIO

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:

0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1
0	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1

1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1
0	0	0	1	1	0	1	1	0	0	0	1	1	0	1	1

ESERCIZIO

```
def es(n):
    sol=[[0]*n for _ in range(n)]
    es1(n,sol)


def es1(n, sol, i=0, j=0):
    if i == n:
        for riga in range(n):
            print(sol[riga])
        print()
        return
    i1, j1= i, j+1
    if j1 == n:
        i1, j1= i+1, 0
    sol[i][j] = 0
    es1(n, sol, i1, j1)
    sol[i][j] = 1
    es1(n, sol, i1, j1)
```

```
>>> es(2)
[0, 0]
[0, 0]

[0, 0]
[0, 1]

[0, 0]
[1, 0]

[0, 0]
[1, 1]

[0, 1]
[0, 0]

[0, 1]
[0, 1]

[0, 1]
[1, 0]

[0, 1]
[1, 1]

[1, 0]
[0, 0]

[1, 0]
[0, 1]

[1, 0]
[1, 0]

[1, 0]
[1, 1]

[1, 1]
[0, 0]

[1, 1]
[0, 1]

[1, 1]
[1, 0]

[1, 1]
[1, 1]
```

ESERCIZIO

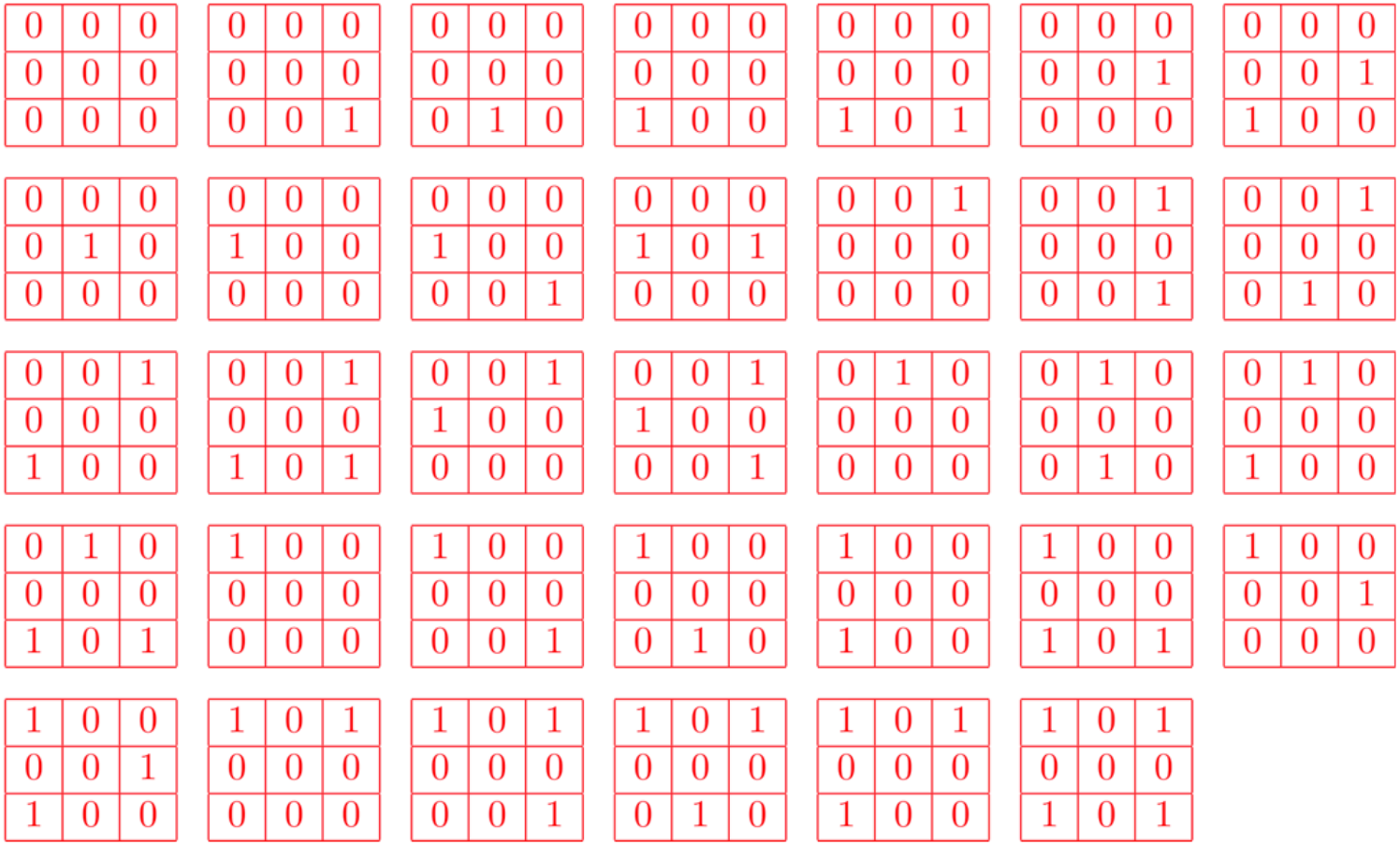
```
def es(n):
    sol=[[0]*n for _ in range(n)]
    es1(n,sol)


def es1(n, sol, i=0, j=0):
    if i == n:
        for riga in range(n):
            print(sol[riga])
        print()
        return
    i1, j1= i, j+1
    if j1 == n:
        i1, j1= i+1, 0
    sol[i][j] = 0
    es1(n, sol, i1, j1)
    sol[i][j] = 1
    es1(n, sol, i1, j1)
```

- 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 $O(n^2)$. L'algoritmo ha complessità $O(2^{n^2}n^2)$.
- Poiché le matrici da stampare sono 2^{n^2} e la stampa di una matrice richiede $\Theta(n^2)$. Qualunque algoritmo per questo problema richiede tempo $\Omega(2^{n^2}n^2)$.
- L'algoritmo è ottimo.

ESERCIZIO

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:



La complessità dell'algoritmo deve essere $O(n^2 S(n))$ dove $S(n)$ è il numero di matrici da stampare.

Algoritmo

- Per la stampa delle matrici eseguiremo una funzione di backtracking.
- nella generica locazione $sol[i][j]$ è potenzialmente consentito inserire 0 o 1. Per ottenere una complessità proporzionale alle $S(n)$ matrici da stampare basta controllare che il bit inserito consenta sempre di ottenere poi una matrice da stampare. Il bit 0 può sempre essere inserito mentre per il bit 1 c'è bisogno di una funzione di taglio che controlli che nelle caselle di sol già riempite e adiacenti a $sol[i][j]$ non sia stato inserito un 1. Le potenziali caselle adiacenti già riempite sono le seguenti 4: $sol[i-1][j-1]$, $sol[i-1][j]$, $sol[i-1][j+1]$ e $sol[i][j-1]$. Inseriamo dunque il simbolo 1 in $sol[i][j]$ solo se

$(i == 0 \ and \ j == 0) \ or$
$(i == 0 \ and \ j > 0 \ and \ sol[i][j-1] == 0) \ or$
$(i > 0 \ and \ j == 0 \ and \ sol[i-1][j] == sol[i-1][j+1] == 0) \ or$
$(i > 0 \ and \ j == n-1 \ and \ sol[i-1][j-1] == sol[i-1][j] == sol[i][j-1] == 0) \ or$
$(i > 0 \ and \ 0 < j < n-1 \ and \ sol[i-1][j-1] == sol[i-1][j] == sol[i-1][j+1] == sol[i][j-1] == 0)$

Implementazione

```
>>> es(2)
[0, 0]
[0, 0]

[0, 0]
[0, 1]

[0, 0]
[1, 0]

[0, 1]
[0, 0]
```

```
def es(n):
    sol=[[0]*n for _ in range(n)]
    es1(n,sol)
```

```
def es1(n, sol, i=0, j=0):
    if i == n:
        for riga in range(n):
            print(sol[riga])
        print()
        return
    i1, j1 = i, j+1
    if j1 == n:
        i1, j1 = i+1, 0
    sol[i][j] = 0
    es1(n, sol, i1, j1)
    if (i==0 and j== 0) or \
    (i==0 and j > 0 and sol[i][j-1]==0) or \
    (i > 0 and j == 0 and sol[i-1][j]==sol[i-1][j+1]==0) or \
    (i > 0 and j == n-1 and sol[i-1][j-1]==sol[i-1][j]==sol[i][j-1]==0) or \
    (i > 0 and 0 < j < n-1 and sol[i-1][j-1]==sol[i-1][j]==sol[i-1][j+1]==sol[i][j-1]==0):
        sol[i][j] = 1
        es1(n, sol, i1, j1)
```

Considerazioni sulla complessità dell'algoritmo

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.

ESERCIZIO

Progettare un algoritmo che prende un intero n e stampa tutte le matrici binarie di dimensioni $n \times n$ in cui righe e colonne risultano ordinate in modo non-decrescente.

Ad esempio per $n = 2$ delle $2^4 = 16$ possibili matrici binarie vanno stampate le seguenti 6 matrici:

0	0
0	0

0	0
0	1

0	0
1	1

0	1
0	1

0	1
1	1

1	1
1	1

La complessità dell'algoritmo deve essere $O(n^2 S(n))$ dove $S(n)$ è il numero di matrici da stampare.

Algoritmo

- Per la stampa delle matrici eseguiremo una funzione di backtracking.
- nella generica locazione $sol[i][j]$ è potenzialmente consentito inserire 0 o 1. Per ottenere una complessità proporzionale alle $S(n)$ matrici da stampare basta controllare che il bit inserito consenta sempre di ottenere poi una matrice da stampare. Il bit 1 può sempre essere inserito mentre per il bit 0 c'è bisogno di una funzione di taglio che controlli che nelle caselle gia riempite di sol adiacenti a $sol[i][j]$ in orizzondate o verticale non sia stato inserito un 1. Le potenziali caselle adiacenti già riempite sono le seguenti 2: $sol[i-1][j]$ e $sol[i][j-1]$. Inseriamo dunque il simbolo 0 in $sol[i][j]$ solo se

$$(i = 0 \ or \ sol[i-1][j] = 0) \ and \ (j = 0 \ or \ sol[i][j-1] = 0)$$

Implementazione

```
def es(n):
    sol=[[0]*n for _ in range(n)]
    es1(n,sol)

def es1(n, sol, i=0, j=0):
    if i == n:
        for riga in range(n):
            print(sol[riga])
        print()
        return
    i1, j1 = i, j+1
    if j1 == n:
        i1, j1 = i+1, 0
    if (i==0 or sol[i-1][j] == 0) and (j== 0 or sol[i][j-1]==0):
        sol[i][j] = 0
        es1(n, sol, i1, j1)
    sol[i][j] = 1
    es1(n, sol, i1, j1)
```

```
>>> es(2)
[0, 0]
[0, 0]

[0, 0]
[0, 1]

[0, 0]
[1, 1]

[0, 1]
[0, 1]

[0, 1]
[1, 1]

[1, 1]
[1, 1]
```

ESERCIZIO

Progettare un algoritmo che prende come parametro l'intero n e stampa tutte le permutazioni dei numeri da 0 a $n-1$.

Ad esempio per $n = 4$ bisogna stampare $4! = 24$ permutazioni:

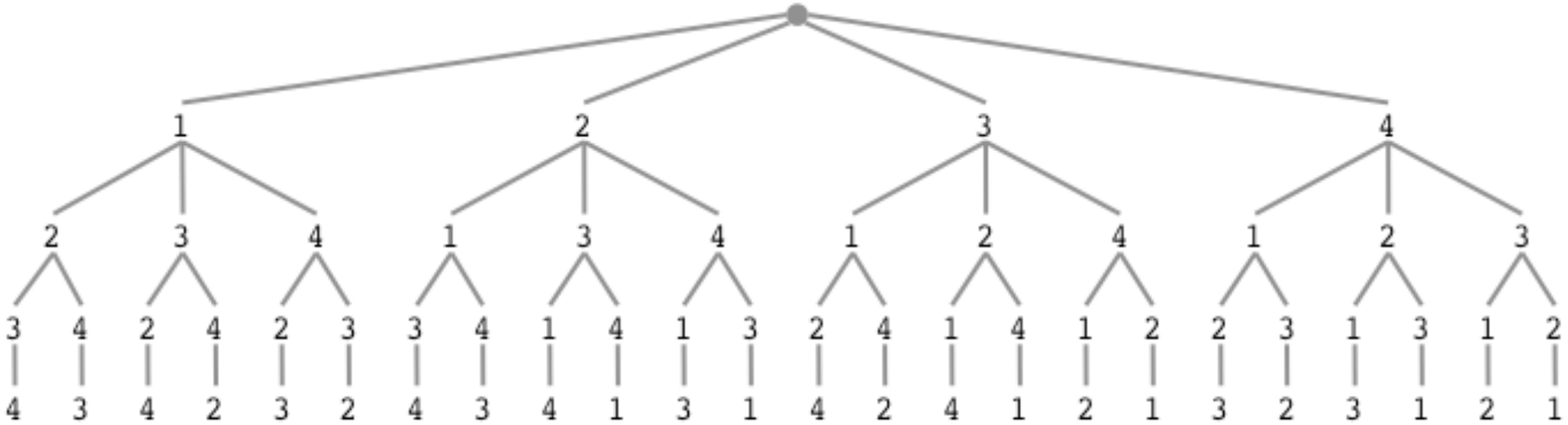
$[0,1,2,3]$ $[0,1,3,2]$ $[0,2,1,3]$ $[0,2,3,1]$ $[0,3,1,2]$ $[0,3,2,1]$ $[1,0,2,3]$

$[1,0,3,2]$ $[1,2,0,3]$ $[1,2,3,0]$ $[1,3,0,2]$ $[1,3,2,0]$ $[2,0,1,3]$ $[2,0,3,1]$

$[2,1,0,3]$ $[2,1,3,0]$ $[2,3,0,1]$ $[2,3,1,0]$ $[3,0,1,2]$ $[3,0,2,1]$ $[3,1,0,2]$

$[3,1,2,0]$ $[3,2,0,1]$ $[3,2,1,0]$

Albero delle permutazioni $(1, 2, 3, 4)$



- L'albero delle permutazioni ha $\Theta(n!)$ foglie e $\Theta(n!)$ nodi interni.
  - nodi interni$= \sum_{i=0}^{n} \frac{n!}{i!} < n! \cdot \sum_{i=0}^{\infty} \frac{1}{i!} \leq n! \cdot \sum_{i=0}^{\infty} \frac{2}{2^i} = n! \frac{2}{1-\frac{1}{2}} = 4 \cdot n!$
  - ho usato
    1. $i! \geq \frac{2^i}{2}$
    2. $\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ per $x < 1$

Algoritmo esaustivo:

```
def es(n):
    preso = [0]*n
    es1(n, [], preso)



def es1(n, sol, preso):
    if len(sol) == n:
        print(sol)
        return
    for j in range(n):
        if preso[j]==0:
            sol.append(j)
            preso[j] = 1
            es1(n, sol, preso)
            sol.pop()
            preso[j] = 0
```

Considerazioni sulla complessità

- L'algoritmo ha complessità $\Theta(n! \cdot n)$
  - l'albero delle permutazioni ha $\Theta(n!)$ nodi interni e $n!$ foglie = $2^{\Theta(n \log n)}$
  - ciascun nodo interno richiede tempo $\Theta(n)$ e ciascuna foglia richiede tempo $\Theta(n)$.
- Qualunque algoritmo per questo problema richiede tempo $\Omega(n! \cdot n)$
  - le soluzioni da stampare sono $n!$ e il tempo per stampare ciascuna di queste è $\Theta(n)$

L'algoritmo proposto è ottimo.

ESERCIZIO

Progettare un algoritmo che prende come parametro l'intero n e stampa tutte le permutazioni dei numeri da 0 a $n-1$ dove nelle posizioni pari compaiono numeri pari.
La complessità dell'algoritmo deve essere $O\left(S(n) \cdot n^2\right)$ dove $S(n)$ è il numero di permutazioni da stampare.

Ad esempio per $n = 5$ delle $5! = 120$ permutazioni bisogna stampare le seguenti 12:

$[0,1,2,3,4]$ $[0,1,4,3,2]$ $[0,3,2,1,4]$ $[0,3,4,1,2]$

$[2,1,0,3,4]$ $[2,1,4,3,0]$ $[2,3,0,1,4]$ $[2,3,4,1,0]$

$[4,1,0,3,2]$ $[4,1,2,3,0]$ $[4,3,0,1,2]$ $[4,3,2,1,0]$

Algoritmo

- Per la stampa delle permutazioni eseguiremo una funzione di backtracking.
- Al passo ricorsivo i-esimo possiamo potenzialmente inserire uno degli n interi non ancora inseriti. Per ottenere una complessità proporzionale alle $S(n)$ permutazioni da stampare la funzione di backtraking controlla che la scelta di inserire un intero non ancora inserito j venga fatta solo se j è pari e la posizione in cui inserire è pari o j è dispari e la posizione in cui inserire è dispari (vale a dire j modulo $2 == i$ modulo 2)

Implementazione

```
def es(n):
    preso = [0]*n
    es1(n, [], preso)


def es1(n, sol, preso):
    if len(sol) == n:
        print(sol)
        return
    for j in range(n):
        if preso[j]==0 and j%2 == len(sol)%2:
            sol.append(j)
            preso[j] = 1
            es1(n, sol, preso)
            sol.pop()
            preso[j] = 0
```

```
>>> es(5)
[0, 1, 2, 3, 4]
[0, 1, 4, 3, 2]
[0, 3, 2, 1, 4]
[0, 3, 4, 1, 2]
[2, 1, 0, 3, 4]
[2, 1, 4, 3, 0]
[2, 3, 0, 1, 4]
[2, 3, 4, 1, 0]
[4, 1, 0, 3, 2]
[4, 1, 2, 3, 0]
[4, 3, 0, 1, 2]
[4, 3, 2, 1, 0]
```

Sia $S(n)$ il numero di permutazioni da stampare.

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

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Esercizio

Dare lo pseudo-codice di un algoritmo che dato l'intero n, stampa tutte le matrici $n \times n$ e valori in $\{a, b, c\}$ con la proprietà che i simboli in ogni riga sono tutti uguali.

Ad esempio per $n = 2$ le matrici da stampare sono:

$$\begin{matrix} a & a \\ a & a \end{matrix} \qquad \begin{matrix} a & a \\ b & b \end{matrix} \qquad \begin{matrix} a & a \\ c & c \end{matrix} \qquad \begin{matrix} b & b \\ a & a \end{matrix} \qquad \begin{matrix} b & b \\ b & b \end{matrix}$$

$$\begin{matrix} b & b \\ c & c \end{matrix} \qquad \begin{matrix} c & c \\ a & a \end{matrix} \qquad \begin{matrix} c & c \\ b & b \end{matrix} \qquad \begin{matrix} c & c \\ c & c \end{matrix}$$

L'algoritmo deve avere complessità $O(n^2 S(n))$ dove $S(n)$ è il numero di matrici da stampare. Motivare la complessità del vostro algoritmo.

Esercizio

Dare lo pseudo-codice di un algoritmo che dato l'intero pari n, stampa tutte le permutazioni dei primi n interi in cui nell'ordinamento non appaiono mai due pari consecutivi nè due dispari consecutivi.
Ad esempio per $n = 4$ devono essere stampate le seguenti 8 permutazioni:

$$1234 \quad 1432 \quad 3214 \quad 3412 \quad 2143 \quad 2341 \quad 4123 \quad 4321$$

L'algoritmo deve avere complessità $O(n^2 S(n))$ dove $S(n)$ è il numero di permutazioni da stampare. Motivare la complessità del vostro algoritmo.

Esercizio

Dare lo pseudocodice di un algoritmo che, dati n e k, con $k \leq n$, stampa tutte le permutazioni dei primi n interi in cui compaiono almeno k punti fissi.

Per una permutazione un punto fisso è un elemento i che compare nella posizione i della permutazione. Ad esempio per n=5 la permutazione $2,1,4,3,0$ contiene due punti fissi (1 e 3).

Ad esempio per $n = 4$ e $k = 2$ il programma deve stampare le seguenti 7 permutazioni:

$$0123,\ 0132,\ 0213,\ 0321,\ 1023,\ 2103,\ 3120$$

La complessità dell'algoritmo deve essere $O(n^2 S(n))$ dove $S(n)$ sono le permutazioni da stampare.