



UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Scienze Matematiche, Fisiche e Naturali
Corso di Laurea in Informatica

Tesi di Laurea

COMPOSIZIONE DI OPERATORI DI
ORDINAMENTO CON CONTENITORI

COMPOSITION OF SORTING OPERATORS
WITH CONTAINERS

ALESSIO SANTORO

Relatore: *Luca Ferrari*

Anno Accademico 2022-2023

RINGRAZIAMENTI

"Può accadere che qualche studioso, al termine della propria fatica, scopra di non dover ringraziare nessuno. Non importa, si inventi dei debiti. Una ricerca senza debiti è sospetta e qualcuno va sempre e in qualche modo ringraziato"

— Umberto Eco

Prima di procedere con la trattazione, vorrei sinceramente dedicare qualche riga a tutti coloro che mi sono stati vicini e, con il loro supporto, mi hanno aiutato in questo meraviglioso percorso.

Nello stendere questa tesi sono stati preziosi i consigli del mio relatore Luca Ferrari, che ha mostrato immensa pazienza ed infinita disponibilità nel guidarmi durante tutto il percorso di stesura dell'elaborato.

Ringrazio di cuore i miei genitori per avermi sempre sostenuto e per avermi permesso di portare a termine gli studi universitari. Un affettuoso grazie va anche a mia sorella Benedetta, che inconsapevolmente ha contribuito molto a farmi arrivare fino a qui.

Non avrei potuto portare a termine questo percorso senza tutti i miei colleghi, compagni di lunghe giornate di studio nel corso di questi anni, tra di loro un grazie speciale va a Lorenzo, Tommaso, Michele e Alessandro. Un ringraziamento speciale va a coloro che mi hanno sostenuto con la loro amicizia da ben prima dell'inizio di questo percorso, crescendo con me e aiutandomi attraverso numerose difficoltà (spero reciprocamente), un abbraccio a Lapo, Chiara, Matteo e Matteo. Una menzione speciale è dovuta a Francesco, che mi ha sempre mostrato affetto e vicinanza.

La mia più profonda gratitudine va infine a Giulia, per il grande supporto, le molte consolazioni e la sua rassicurante presenza nei momenti di sconforto. Grazie per tutto il tempo che mi hai dedicato. Grazie perchè ci sei sempre stata.

INDICE

1	Introduzione	5
2	Algoritmi di ordinamento e classi di pattern	7
2.1	Algoritmi	8
2.1.1	Bubblesort	8
2.1.2	Stacksort	8
2.1.3	Queuesort	9
2.2	Classi di pattern di permutazioni	11
2.3	Algoritmi di ordinamento e classi di pattern	14
3	Programmi realizzati	15
3.1	PermutaSort	15
3.2	PattFinder	17
3.3	Script per la verifica dei pattern	19
4	Composizioni di operatori di ordinamento	23
4.1	Algoritmo per le controimmagini secondo bubblesort	24
4.2	Composizioni di un operatore di ordinamento con bubble- sort	25
4.2.1	Esempio pratico di calcolo di una controimmagine: composizione di stacksort con bubblesort	25
4.2.2	Composizione di queuesort con bubblesort	27
4.3	Composizioni di un operatore di ordinamento con stack- sort	27
4.3.1	Composizione di queuesort con stacksort	28
4.3.2	Composizione di bubblesort con stacksort	29
4.4	Composizioni di un operatore di ordinamento con <i>queue- sort</i>	29
4.4.1	Composizione di stacksort con queuesort	30
4.4.2	Composizione di bubblesort con queuesort	31
4.5	Contenitori POP	32
4.5.1	POP-Queuesort - dettagli ulteriori	32
4.6	Composizioni che includono operatori POP	34
4.6.1	Composizione di POP-stacksort con stacksort	34
4.6.2	Composizione di POP-stacksort con bubblesort	35
4.6.3	Composizione di POP-stacksort con queuesort	35
4.6.4	Composizione di POP-queuesort con stacksort	36
4.6.5	Composizione di POP-queuesort con bubblesort	38

4 Indice

4.6.6	Composizione di POP-queuesort con queuesort	40
4.6.7	Presentazione dell'algoritmo per la ricerca di controimmagini secondo l'operatore POPstacksort	41
4.6.8	Osservazione sulle analisi di composizioni di operatori di ordinamento con POP-queuesort	42
4.6.9	Composizione di stacksort con POP-queuesort	43
4.6.10	Composizione di queuesort con POP-queuesort	47
4.6.11	Composizione di bubblesort con POP-queuesort	48
5	Conclusioni	51

INTRODUZIONE

Gli algoritmi di ordinamento sono uno degli argomenti più classici dell'informatica.

In questa tesi si esamina una classe di questi algoritmi, ovvero quelli che utilizzano dei contenitori: strutture dati in cui vengono salvati alcuni elementi della sequenza da ordinare. Il più noto tra questi algoritmi è bubblesort.

Nel caso di bubblesort, il primo elemento della sequenza in input viene salvato in un contenitore apposito di capacità 1, quindi si continua a scorrere la sequenza; ad uno ad uno si confrontano gli elementi dell'input con l'elemento contenuto nel contenitore, aggiungendo all'output il minore dei due mentre il maggiore viene salvato nel contenitore.

Due ulteriori algoritmi, stacksort e queuesort, si ottengono generalizzando il contenitore, in particolare sostituendolo con una struttura dati (rispettivamente una pila e una coda) di capacità arbitraria.

Gli algoritmi presentati generalmente vengono iterati più volte sulla stessa sequenza. La ragione è che una sola iterazione non garantisce l'ordinamento completo della sequenza.

In questa tesi tuttavia si osserva il caso dell'applicazione di una sola iterazione di questi algoritmi ad una permutazione e delle proprietà che permettono di stabilire per ogni algoritmo l'insieme esatto di permutazioni ordinabili da una sola iterazione.

Questo tema è stato inizialmente trattato da Donald Knuth, in "Vol. 1: Fundamental Algorithms", *The Art of Computer Programming* (1968), che aveva individuato esattamente quali permutazioni sono ordinabili da una sola passata di stacksort. Similmente sono state trovate condizioni analoghe per gli altri algoritmi.

L'obiettivo di questa tesi è quello di utilizzare le condizioni note per

ricercare le controimmagini di determinati pattern di permutazioni di interi per individuare condizioni che garantiscono l'ordinabilità di una permutazione da parte della composizione di due algoritmi tra quelli presentati.

Per trovare questi risultati si utilizzano degli algoritmi che permettono di ricercare le controimmagini di pattern di permutazioni secondo gli algoritmi in esame, presenti in letteratura.

Ci si è avvalsi dell'utilizzo di strumenti software, realizzati appositamente, per esaminare il comportamento di questi algoritmi: uno che enumera le permutazioni ordinabili e non ordinabili da un dato algoritmo di ordinamento, insieme alle possibili immagini prodotte, e uno che mostra quali permutazioni contengano o meno un dato pattern.

Nel capitolo 2 di questa tesi vengono introdotti gli algoritmi di ordinamento in esame, nonché alcuni concetti legati alle classi di pattern di permutazioni utilizzati nel corso della trattazione.

Nel capitolo 3 vengono presentati i programmi realizzati e ne viene spiegato l'utilizzo. Il codice sorgente di tali programmi è riportato interamente nell'appendice finale.

Nel capitolo 4 vengono mostrati i procedimenti per individuare le classi di pattern ordinabili dalle composizioni; vengono introdotti gli algoritmi per il calcolo di controimmagini e ne viene mostrata l'applicazione.

Nel capitolo finale si riassumono i risultati trovati e si mostra, per ogni composizione, la quantità di permutazioni ordinabili di lunghezza n .

ALGORITMI DI ORDINAMENTO E CLASSI DI PATTERN

In questo capitolo verranno introdotti alcuni tra i algoritmi di ordinamento che utilizzano opportune strutture dati come contenitori intermedi per gli elementi della permutazione in input, in particolare *stacksort*, *queuesort* e *bubblesort*, e altri concetti necessari per l'analisi della composizione tra due o più esecuzioni di tali algoritmi.

Durante la loro esecuzione questi algoritmi possono salvare gli elementi in un contenitore (la diversa struttura dati adottata definisce i diversi algoritmi) dalla quale poi vengono prelevati per essere aggiunti all'output.

In questa tesi gli algoritmi prendono in input permutazioni di interi, sebbene sia possibile estendere la trattazione a parole arbitrarie su un insieme totalmente ordinato (anche se non sempre in modo immediato). Per rappresentare le permutazioni si usa la notazione lineare, quindi una permutazione π di lunghezza n è una parola sull'alfabeto $\{1, 2, \dots, n\}$ in cui ogni lettera compare una e una sola volta.

INVERSIONI In una permutazione $\pi = \pi_1\pi_2 \dots \pi_n$ un'inversione è una coppia di valori (π_i, π_j) , dove $i < j, \pi_i > \pi_j$. Una coppia di valori che non compone una inversione si dice una non inversione.

Una singola iterazione degli algoritmi che prenderemo in considerazione non garantisce l'ordinamento della permutazione in input, dunque gli algoritmi devono essere iterati più volte, ogni volta sul risultato della iterazione precedente. In ogni caso alla fine della i -esima iterazione i maggiori i elementi avranno raggiunto la loro posizione finale, dunque sono necessarie al massimo $n - 1$ iterazioni per ordinare la permutazione.

In questo capitolo si mostrerà il comportamento di una singola applica-

zione di questi algoritmi. Le proprietà trovate verranno poi sfruttate per lo studio di varie composizioni.

Infatti generalmente quando si parla di questi algoritmi essi vengono descritti in funzione di varie iterazioni che vengono applicate fino a che la permutazione non è ordinata, mentre qui esamineremo gli effetti di una singola iterazione.

Ad esempio prendendo l'algoritmo bubblesort (certamente il più noto fra quelli che verranno presentati) si farà riferimento ad un operatore B definito appositamente: dove π è una permutazione di interi, $B(\pi)$ rappresenta il risultato di una sola iterazione di bubblesort su di essa.

2.1 ALGORITMI

2.1.1 *Bubblesort*

L'algoritmo di ordinamento bubblesort prevede di scorrere dal primo al penultimo elemento e confrontare ciascuno con il proprio successivo per scambiarli se non sono ordinati.

Si definisce l'operazione $\text{Swap}(\pi_i, \pi_j)$, che fissata una permutazione π la restituisce con l' i -esimo e il j -esimo elementi scambiati di posizione.

Il risultato di una singola iterazione di bubblesort su una permutazione $\pi = \pi_1\pi_2 \dots \pi_n$ è calcolato dall'operatore B e si indica con $B(\pi)$.

Si può scrivere permutazione π come $\pi_L n \pi_R$, dove n è il massimo valore della permutazione, π_L è il suo prefisso e π_R il suo suffisso. Vale che $B(\pi) = B(\pi_L)\pi_R n$.

Algorithm 1 B – bubblesort

```

1: for  $i = 1$  to  $n - 1$  do
2:   if  $\pi_i > \pi_{i+1}$  then
3:      $\text{Swap}(\pi_i, \pi_{i+1})$ 
4:   end if
5: end for
```

2.1.2 *Stacksort*

Stacksort è un algoritmo che utilizza una pila per ordinare una permutazione.

Si definisce una pila come un insieme di interi su cui si possono applicare le seguenti operazioni:

$\text{Push}(\pi_i, S)$ aggiunge l'elemento π_i alla pila S

$\text{Pop}(S)$ estrae dalla pila S l'ultimo intero inserito e lo aggiunge all'output

$\text{Top}(S)$ permette di osservare il valore dell'elemento che verrebbe estratto da $\text{Pop}(S)$ senza rimuoverlo.

Per ogni elemento π_i presente nell'input se la pila è vuota viene eseguito un push e si passa al prossimo. Se la pila non è vuota si esegue l'operazione pop fino a che l'elemento in cima alla pila non è maggiore di π_i , poi si esegue un push.

Finiti gli elementi nell'input, se necessario, si svuota completamente la pila nell'output.[8]

L'espressione $S(\pi)$ rappresenta il risultato ottenuto applicando un'iterazione di stacksort su una permutazione $\pi = \pi_1\pi_2 \dots \pi_n$.

In questo caso, sia $\pi = \pi_L n \pi_R$, con n valore massimo in π , e π_L, π_R rispettivamente suo prefisso e suffisso, vale che $S(\pi) = S(\pi_L)S(\pi_R)n$.

Algorithm 2 S - stacksort

```

1:  $S \leftarrow \emptyset$ 
2: for  $i = 1$  to  $n$  do
3:   while  $S \neq \emptyset$  and  $\pi_i > \text{Top}(S)$  do
4:      $\text{Pop}(S)$ 
5:   end while
6:    $\text{Push}(\pi_i, S)$ 
7: end for
8: while  $S \neq \emptyset$  do
9:    $\text{Pop}(S)$ 
10: end while
```

2.1.3 Queuesort

Queuesort utilizza una coda per ordinare una permutazione.

Si definisce una coda come un insieme su cui si possono applicare le seguenti operazioni:

$\text{Enqueue}(\pi_i, Q)$ aggiunge l'elemento π_i alla coda Q

Dequeue(Q) estrae da Q l'elemento che è stato inserito per primo e lo aggiunge all'output

Front(Q) permette di osservare del prossimo valore da estrarre senza rimuoverlo dalla pila

Back(Q) permette di osservare l'ultimo valore inserito nella pila.

Per descrivere queuesort, è infine necessaria un'ultima operazione che non prevede di utilizzare la coda:

Bypass(π_i) il valore π_i viene aggiunto all'output senza passare dalla coda

Per ogni elemento π_i della permutazione π in input se la coda è vuota o il suo ultimo elemento è minore di π_i , si accoda π_i , altrimenti si tolgono elementi dalla coda ponendoli nell'output fino a che l'elemento in testa alla coda non è maggiore di π_i , poi si aggiunge π_i all'output. Si svuota la coda nell'output.

Algorithm 3 operatore Q - queuesort, singola iterazione

```

1:  $Q \leftarrow \emptyset$ 
2: for  $i = 1$  to  $n$  do
3:   if  $Q = \emptyset$  or  $\text{Back}(Q) < \pi_i$  then
4:     Enqueue( $\pi_i, Q$ )
5:   else
6:     while  $\text{Front}(Q) < \pi_i$  do
7:       Dequeue( $\pi_i$ )
8:     end while
9:     Bypass( $\pi_i$ )
10:  end if
11: end for
12: while  $Q \neq \emptyset$  do
13:   Dequeue( $\pi_i$ )
14: end while

```

OSSERVAZIONE *bubblesort* è un caso particolare sia di *queuesort* che di *stacksort*.

Se infatti si fissa a 1 la capacità della pila o della coda dei rispettivi operatori il comportamento che questi assumono è quello di una cella che, scorrendo l'input, contiene sempre il massimo valore trovato, mentre gli altri vengono messi nell'output, ovvero analogo a quello di *bubblesort*.

LEFT-TO-RIGHT MASSIMI In una permutazione i *LTR massimi* sono gli elementi che risultano essere maggiori di ogni altro elemento che li precede. Ad esempio nella permutazione 142387596 i *LTR massimi* sono 1,4,8,9.

La nozione di *LTR massimi* risulta molto importante per lo studio di alcuni degli algoritmi di ordinamento presentati. In particolare durante l'esecuzione di *queuesort* i LTR massimi sono tutti e soli gli elementi che entrano nella coda, mentre durante l'esecuzione di *bubblesort* sono gli unici che vengono scambiati; in entrambi gli algoritmi l'ordine relativo degli altri elementi non viene alterato.

CONTENITORI POP Un caso di studio interessante è quello in cui i contenitori di *stacksort* e *queuesort* vengano sostituiti dalla loro versione POP, ovvero un contenitore con politiche di estrazione e inserimento analoghe ma quando viene eseguita un'estrazione il contenitore viene svuotato completamente. Si definiscono con questa variante, gli algoritmi **pop-stacksort** e **pop-queuesort**

2.2 CLASSI DI PATTERN DI PERMUTAZIONI

Siano α, β due sequenze di interi indichiamo con $\alpha \subseteq \beta$ che α è una sottosequenza di β , anche se non necessariamente una sottosequenza consecutiva[2].

STANDARDIZZAZIONE Un pattern classico è sempre rappresentato dalla **standardizzazione** di una permutazione.

Per una sequenza di numeri la sua standardizzazione[7] è un'altra sequenza della stessa lunghezza in cui l'elemento minore della sequenza originale è stato sostituito da 1, il secondo minore con un 2, ecc.

Ad esempio, la standardizzazione di 5371 è 3241 .

Si dice che una permutazione τ contiene un pattern δ se esiste $\lambda \subseteq \tau$ la cui standardizzazione è uguale a δ , e si indica con $\delta \preceq \tau$.

Ad esempio 24153 contiene il pattern 312 perché $413 \subseteq 24153$ e la standardizzazione di 413 è 312.

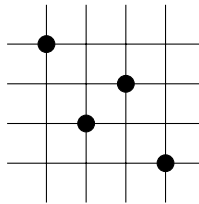
CLASSI DI PATTERN La relazione di sottopermutazione è una relazione d'ordine parziale che viene studiata con dei sottoinsiemi chiamati

pattern di classi. Ogni classe D può essere caratterizzata dall'insieme minimo M che evita:

$$D = Av(M) = \{\beta : \mu \not\leq \beta \forall \mu \in M\}$$

In questa sezione verranno da qui in poi introdotti brevemente altri tipi di pattern[3] attraverso degli esempi; le macro per la realizzazione delle griglie sono state prodotte dalla *Reykjavik University*[10].

PLOT, RAPPRESENTAZIONE GRAFICA DI PERMUTAZIONI Data una permutazione $\pi = \pi_1\pi_2 \dots \pi_n$ la sua rappresentazione grafica è data dall'insieme di punti (i, π_i) . Questi punti possono essere rappresentati in un piano cartesiano, in particolare nella griglia sottostante, che rappresenta il pattern 4231, le righe verticali rappresentano le ascisse i (numerate a partire da 1 da quella più a sinistra) e quelle orizzontali le ordinate, quindi i valori interi π_i (numerati a partire da 1 da quella più in basso).

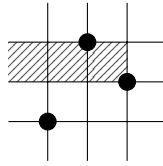


PATTERN BARRATI I pattern classici esprimono in quale relazione d'ordine devono essere gli elementi di una sequenza. In alcuni casi può rivelarsi necessario dover esprimere informazioni ulteriori, come l'assenza di un elemento in una data posizione. Questa informazione può essere fornita grazie all'utilizzo dei pattern barrati.

Un pattern barrato è un pattern in cui ogni numero può essere barrato. Un'istanza di un pattern barrato è rappresentata da un'istanza della parte non barrata del pattern che non si estenda ad un'occorrenza della parte barrata. Ad esempio una permutazione π contiene il pattern $2\bar{3}41$ se contiene degli elementi $a < b < c$ tali che $\pi = \dots b \dots c \dots a \dots$ e fra c e a non è presente un elemento maggiore di c .

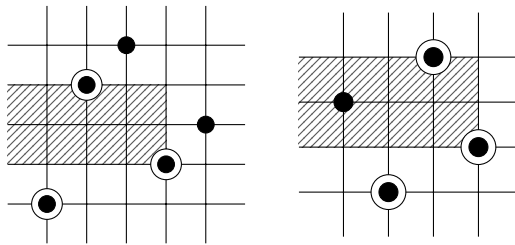
Una permutazione quindi evita un pattern barrato se ad ogni occorrenza della parte non barrata del pattern corrisponde un'occorrenza della parte barrata.

MESH PATTERN I pattern barrati possono essere rappresentati da plot come quello mostrato prima, dove l'area corrispondente all'elemento barrato viene oscurata. Si osservi il seguente pattern a titolo di esempio:

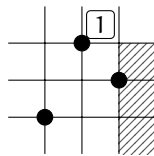


Una permutazione π di lunghezza n contiene questo pattern se ci sono 3 valori $a < b < c < n$ tali che $\pi = \dots a \dots c \dots b \dots$ e prima di b nessun valore è compreso tra b e c .

Ad esempio la permutazione 14523 (mostrata sotto a sinistra) contiene un'istanza di questo pattern mentre la permutazione 3142 (mostrata a destra) no. In entrambi i casi l'occorrenza della parte non barrata del pattern è evidenziata.



PATTERN DECORATI Un pattern decorato è un mesh-pattern in cui vengono aggiunte informazioni relative alla presenza di elementi in determinate aree. Senza dare la definizione formale descriviamo un esempio di tale pattern grazie al quale introduciamo anche la notazione che verrà utilizzata. Nel seguente pattern l'area bianca indicata con 1 indica che per avere un'istanza di questo pattern in una permutazione, all'interno dell'area deve essere presente almeno un altro elemento della permutazione:



Questo pattern è contenuto, ad esempio, nelle sequenze 1342, 12453 o 13524.

Talvolta è possibile fornire informazioni più complesse della quantità di elementi presenti in una area, tipo l'ordine relativo che tali elementi assumono tra di loro. Si mostrerà più avanti come alcuni pattern decorati possano essere utilizzati per mostrare la presenza di inversioni o non inversioni all'interno di un pattern.

Se una decorazione è oscurata questa indica una permutazione contiene un'occorenza del pattern se nella zona occupata dalla decorazione non sono presenti elementi corrispondenti alle informazioni fornite dalla decorazione.

2.3 ALGORITMI DI ORDINAMENTO E CLASSI DI PATTERN

Per tutti gli algoritmi presentati all'inizio del capitolo sono già note le classi di pattern che garantiscono l'ordinabilità con una sola passata.

Operatore	Permutazioni ordinabili con una sola passata
Stacksort[8]	$Av(231)$
Queuesort[6]	$Av(321)$
Bubblesort[1]	$Av(231, 321)$
Pop-stacksort[9]	$Av(231, 312)$
Pop-queuesort[4]	$Av(321, 2413)$

PROGRAMMI REALIZZATI

In questo capitolo si presenta il funzionamento di alcuni programmi realizzati come supporto allo studio degli argomenti di questa tesi.

Tutto il software che verrà presentato è stato realizzato con il linguaggio Python, nel seguente ambiente: 3.11.6 (main) [GCC 13.2.1 20230801]. Tutti gli script vengono lanciati da linea di comando e mostrano i risultati sulla console.

3.1 PERMUTASORT

Il primo programma che ho realizzato viene lanciato da linea di comando specificando come argomenti un intero n e un operatore di ordinamento X . Il suo scopo è quello di enumerare tutte le n -permutazioni ordinabili e non ordinabili con una sola passata di X , stampando anche i possibili risultati di X su n -permutazioni.

La classe `SelectorPermutations` è il core del programma, viene inizializzata passando un intero e un operatore al costruttore. Grazie alla funzione `itertools.permutations` vengono generate le n -permutazioni e, scorrendole tutte, vi si applica l'operatore X . Il risultato viene aggiunto alla lista `outcomes` e viene valutato: se questo è ordinato la relativa permutazione da cui si è ottenuto viene aggiunta alla lista `sortable`, altrimenti a `unsortable`.

La classe fornisce dei *getters* per le liste e non ha altri metodi, oltre al costruttore.

Dopo essere stata istanziata la classe viene usata come riferimento per i dati che ha già calcolato nel costruttore, e non produce ulteriori risultati. Di seguito viene mostrato il listato della classe:

```

class selectorPermutations:
    def __init__(self,num,op):
        if num <=0:
            print("ERROR: was expecting a positive integer but got " +str(num))
            exit()

        # initialization of lists
        self.__sortable = []
        self.__unsortable = []
        self.__outcomes = []

        # generation of permutations
        permutations_list = list(permutations(range(1,num+1)))

        for P in permutations_list:

            # applying the operator to the permutation
            op_P_ = op(P)

            # adding outcome to the list
            if op_P_ not in self.__outcomes:
                self.__outcomes.append(op_P_)

            # adding permutations to the right list
            if isIdentityPermutation(op_P_):
                self.__sortable.append(P)
            else:
                self.__unsortable.append(P)

    def getSortable(self):
        return self.__sortable

    def getUnsortable(self):
        return self.__unsortable

    def getOutcomes(self):
        return self.__outcomes

```

3.2 PATTFINDER

Questo programma ha lo scopo di selezionare tra tutte le permutazioni di una data dimensione quali contengono un dato pattern classico e quali no.

Il programma viene lanciato da linea di comando prendendo come argomenti un intero e una sequenza di interi consecutivi eventualmente non ordinati.

Il primo intero rappresenta la lunghezza delle permutazioni da scorrere e la sequenza rappresenta il pattern da ricercare.

Gli argomenti vengono passati al costruttore della classe `PatternAvoid`, l'intero `n` viene utilizzato per generare le `n`-permutazioni grazie alla funzione `permutations`. Per ogni permutazione viene controllato se contiene o no il pattern e viene inserita nella rispettiva lista: `containing` o `notcontaining`.

La classe presenta i getter per le liste prodotte e due ulteriori metodi:

`patternize` serve a *standardizzare* una sequenza, si fa riferimento alla definizione di standardizzazione (di una permutazione) presentata nel capitolo precedente

`contains` verifica se una sequenza contiene un pattern o meno, grazie alla funzione `itertools.combinations` si ottengono tutte le sottosequenze (anche non consecutive), le si standardizzano e le si confrontano con il pattern da ricercare.

```
class PatternAvoid():
    def __init__(self, n, pattern):
        if n <= 1:
            print("ERROR: expecting a whole number greater than 1 but got: "
                  + str(n))
            exit()

        self.__n = n

        # checking if the argument really is a classical pattern
        pattern_list = [int(p) for p in pattern]
        if len(pattern) != max(pattern_list):
            print("ERROR: expected a pattern but got: " + str(pattern))
            exit()
        for i in range(1, len(pattern)):
```

```

        if i not in pattern_list:
            print("ERROR: expected a pattern but got: " + str(pattern))
            print("\nA pattern of length n must have all the numbers from 1
                to n, " + str(i) + " not present")
            exit()

# generating permutations
permutations_list = list(permutations(range(1,n+1)))

# declaring list fields
self.__notcontaining = []
self.__containing = []

for p in permutations_list:
    if not self.contains(p,pattern):
        self.__notcontaining.append(p)
    else:
        self.__containing.append(p)

# get the standard image of the number sequence:
def patternize(self,pi):
    output = []
    id_ = sorted(pi)
    for p in pi:
        output.append(id_.index(p)+1)
    return output

# check if a sequence contains the given pattern
def contains(self, seq, pattern):
    if len(seq) < len(pattern):
        return False
    subseq = combinations(seq, len(pattern))
    for s in subseq:
        if list(self.patternize(s)) == list(pattern):
            return True
    return False

def getNotContaining(self):
    return self.__notcontaining

def getContaining(self):
    return self.__containing

```

3.3 SCRIPT PER LA VERIFICA DEI PATTERN

Spesso si è utilizzato un ulteriore script, che si serve delle classi presentate prima, per verificare che i pattern trovati per un dato operatore siano sufficienti a definire tutto l'insieme di permutazioni non ordinabili.

Grazie a `selectorPermutations` vengono generate tutte le permutazioni non ordinabili (l'operatore e la dimensione delle permutazioni vengono passate come argomenti allo script) ed in seguito si scorre la lista di pattern che si vogliono verificare. Per ogni pattern `PatternAvoid` genera le permutazioni che lo contengono e queste vengono rimosse dalla lista di permutazioni non ordinabili. Se alla fine la lista rimane vuota allora viene stampato a schermo che tutte le permutazioni non ordinabili contengono almeno uno dei pattern specificati, altrimenti elenca quelle rimaste.

Lo script viene lanciato da linea di comando con i seguenti argomenti (nell'ordine): l'operatore di ordinamento, la dimensione delle permutazioni, l'elenco di pattern da verificare.

Di seguito si presenta un esempio di utilizzo ed il codice sorgente dello script; l'utilizzo mostrato è in riferimento alla combinazione degli operatori `stacksort` e `bubblesort`, che verrà spiegata nel dettaglio nel capitolo successivo.

```
$ python verify_patterns.py SB 5 2341 2431 3241
```

```
Generating permutations:
```

```
    found 50 unsortable permutations
```

```
Verifying 3 patterns:
```

```
Verifying pattern 2341:
```

```
    found 17 matches
```

```
Verifying pattern 2431:
```

```
    found 17 matches
```

```
Verifying pattern 3241:
```

```
    found 17 matches
```

These 10 permutations are not sorted by SB but not contain any specified pattern:

```
(5, 2, 4, 1, 3)
(5, 3, 1, 4, 2)
(5, 2, 3, 1, 4)
(5, 3, 4, 2, 1)
(1, 5, 3, 4, 2)
(5, 1, 3, 4, 2)
(4, 5, 2, 3, 1)
(4, 2, 3, 1, 5)
(5, 4, 2, 3, 1)
(5, 3, 4, 1, 2)
```

```
$ python verify_patterns.py SB 5 2341 2431 3241 4231
```

```
Generating permutations:
    found 50 unsortable permutations
```

```
Verifying 4 patterns:
```

```
Verifying pattern 2341:
    found 17 matches
```

```
Verifying pattern 2431:
    found 17 matches
```

```
Verifying pattern 3241:
    found 17 matches
```

```
Verifying pattern 4231:
    found 17 matches
```

```
All unsortable permutations contains some of the specified patterns
```

```
import sys
from permutasort.permutasort import *
from pattFinder.pattfinder import *

# allows to find out if the specified patterns cover all the
# permutations of lenght n
# that are not sortable by the specified sorting operator
```

```

if __name__ == '__main__':

    if len(sys.argv) < 4:
        print("ERROR; requested at least 3 arguments but got: " +
              str(sys.argv[1:]))
        exit()

    op = getOperator(sys.argv[1])
    n = int(sys.argv[2])
    av = sys.argv[3:]

    print("Generating permutations: ")
    unsortable = selectorPermutations(n,op).getUnsortable()
    result = unsortable
    print("\tfound " + str(len(unsortable)) + " unsortable
          permutations\n")

    print("Verifying "+str(len(av))+" patterns:\n")
    for p in av:
        print("Verifying pattern " + p+":")
        # from the list of unsortable permutations the ones containing
        # the specified patterns get removed
        pattern = [int(char) for char in p]
        contains_p = PatternAvoid(n, pattern).getContaining()
        result = list(set(result).difference(set(contains_p)))
        print("\tfound
              "+str(len(set(contains_p).intersection(set(unsortable)))) +
              " matches\n")

    n_uns = len(result)
    if(n_uns==0):
        print("All unsortable permutations contains some of the specified
              patterns\n")
    else:
        print("These "+str(n_uns)+" permutations are not sorted by " +
              sys.argv[1]+" but not contain any specified pattern:")
        print(printlist(result))

```

COMPOSIZIONI DI OPERATORI DI ORDINAMENTO

Da questo momento si mostreranno le varie composizioni di operatori e le relazioni tra relative classi di pattern e le loro controimmagini.

CONTROIMMAGINI Sia X un operatore di ordinamento, la controimmagine di una certa permutazione p , secondo X , indicata con $X^{-1}(p)$ rappresenta l'insieme di tutte le permutazioni la cui immagine secondo l'operatore X è uguale a p .

$$X^{-1}(p) = \{\beta : p = X(\beta)\}$$

$Av(21)$ è l'insieme formato dalle sole permutazioni identità, dato che contiene tutte le permutazioni in cui nessun elemento sia disordinato rispetto ad un altro, ovvero solo le permutazioni crescenti. Sia X un operatore di ordinamento si indica con $X^{-1}(Av(21))$ l'insieme di tutte le permutazioni ordinabili da X .

Ad esempio si scrive $B^{-1}(Av(21)) = Av(231, 321)$ per indicare che l'insieme di permutazioni ordinabili dall'operatore bubblesort coincide con l'insieme di permutazioni che evitano il pattern $231, 321$ [7].

Siano X, Y due operatori tali che sia noto l'insieme $X^{-1}Av(21) = Av(M)$, dove M è un insieme minimo di pattern, allora l'insieme di permutazioni ordinabili dalla loro composizione XY è dato da:

$$X \circ Y^{-1}(Av(21)) = Y^{-1}X^{-1}(Av(21)) = Y^{-1}Av(M)$$

Dunque conoscendo quali pattern debbano essere evitati per garantire l'ordinabilità da parte di X si possono ricercare le controimmagini di quegli stessi pattern secondo Y per avere l'insieme ordinabile dalla loro composizione.

Ogni analisi delle composizioni mostrate in questa tesi inizierà con un espressione che mostri quindi quali controimmagini e secondo quale operatore è necessario cercare.

ALGORITMI PER IL CALCOLO DI CONTROIMMAGINI Sono già stati prodotti alcuni algoritmi per calcolare le controimmagini di pattern secondo gli operatori *bubblesort*[1], *stacksort*[7], *queuesort*[9] e *POP-stacksort*[9], che verranno esposti in seguito e utilizzati per l'analisi delle composizioni.

UN CASO GIÀ NOTO: COMPOSIZIONE DI STACKSORT CON BUBBLESORT
Si considera dunque la composizione $SB = S \circ B$ e ci si chiede quali permutazioni possano essere ordinate da essa.

$$(S \circ B)^{-1}Av(21) = B^{-1}S^{-1}(Av(21)) = B^{-1}(Av(231))$$

Dunque ricercando per quali permutazioni *bubblesort* evita il pattern 231 si trovano le condizioni per cui una permutazione risulta ordinabile da SB .

Utilizzando l'algoritmo per le controimmagini di *bubblesort*[1] si ottiene il seguente risultato, che verrà verificato in seguito:

$$(S \circ B)^{-1}(Av(21)) = Av(3241, 2341, 4231, 2431)$$

4.1 ALGORITMO PER LE CONTROIMMAGINI SECONDO BUBBLESORT

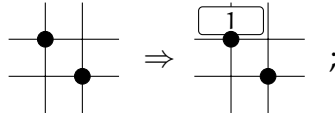
Per ottenere una controimmagine di *bubblesort* di un pattern classico, si applicano le seguenti regole:

- si cercano tutte le inversioni nell'immagine in esame: poiché *bubblesort* non produce nuove inversioni, tutte le inversioni nell'immagine dovranno già essere presenti nella controimmagine;

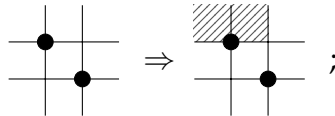
- si considera una lista di **candidati**, composta da ogni pattern minimale che contiene almeno le stesse inversioni;

- per ogni candidato, si processa ogni sua inversione (b, a) nel modo seguente:

- se (b, a) è contenuta anche nell'immagine si ha che b non è un LTR massimo o che lo è ma c'è un altro LTR massimo tra b e a , quindi si decora la coppia come segue, producendo un nuovo pattern:



se (b, a) è un'inversione del candidato che non è nell'immagine allora la coppia viene ordinata da una passata di bubblesort, si ha che b è l'ultimo LTR massimo prima di a , la coppia viene decorata nel modo seguente:



applicare ad un candidato le condizioni per tutte le sue inversioni (considerando che se una zona è oscurata da una condizione, non può essere decorata da un'altra) fornisce un mesh-pattern decorato che descrive il pattern contenuto nella controimmagine.

4.2 COMPOSIZIONI DI UN OPERATORE DI ORDINAMENTO CON BUBBLESORT


4.2.1 Esempio pratico di calcolo di una controimmagine: composizione di stacksort con bubblesort

Prima di tutto si ricerca di quale classe di pattern nonché tramite quale operatore occorre calcolare le controimmagini:

$$(S \circ B)^{-1}(Av(21)) = B^{-1}(S^{-1}(Av(21))) = B^{-1}(Av(231))$$

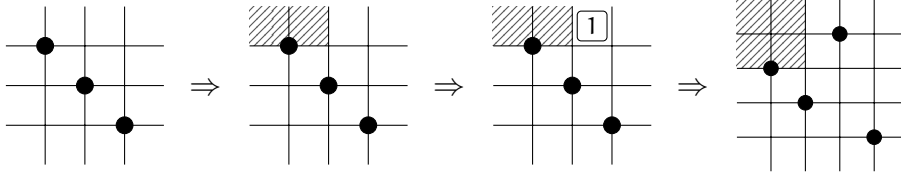
Si ricorda che vale che $S^{-1}Av(21) = Av(231)$. I pattern candidati ad essere controimmagini di 231 sono i pattern minimali che contengono le coppie $(2, 1)$, $(3, 1)$, ovvero i pattern $231, 321$. Si esaminano separatamente i due candidati.

231: si osserva che una condizione per l'inversione $(2, 1)$ è già realizzata dall'elemento 3, applicando le regole descritte prima all'inversione $(3, 1)$

si ha il mesh-pattern , evitare quest'ultimo equivale ad evitare contemporaneamente i pattern classici $2341, 2431, 4231$.

321: in questo caso si ha l'inversione $(3, 2)$ che deve essere invertita, quindi si inseriscono le aree oscurate, in seguito si osserva che, come prima, 3 realizza la condizione per l'inversione $(2, 1)$ e dunque basta

aggiungere le decorazioni per l'inversione $(3, 1)$ tenendo conto delle aree oscurate. Di seguito si mostrano i passaggi:



Il pattern finale è simile ad un pattern classico 3241, presentando in più delle aree oscurate. Si può tuttavia osservare come un eventuale elemento contenuto in quelle aree oscurare può solo formare un ulteriore pattern 3241 (assumendo il ruolo del 3) o un pattern 4231 con gli ultimi 3 elementi del pattern.

In ultima analisi dunque evitare il mesh-pattern trovato, tenendo anche conto degli altri pattern trovati, equivale ad evitare il pattern classico 3241 che, unito agli altri fornisce esattamente il risultato atteso $(SB)^{-1}(Av(21)) = Av(2341, 2431, 4231, 3241)$.

Si osservi che il software `permutasort`, presentato nel capitolo 3, può aiutare ad arrivare alle stesse conclusioni. Infatti lanciando il programma sull'operatore `SB` e con $n = 4$ si ottengono i seguenti risultati:

```
$ python permutasort.py 4 SB
...
The following 4 4-permutations are not sortable with the operator
SB:
(2, 3, 4, 1)
(2, 4, 3, 1)
(3, 2, 4, 1)
(4, 2, 3, 1)
...
```

Questo risultato è utile come indicazione, ma non fornisce garanzia di aver trovato i pattern corretti: infatti non si può escludere la presenza di pattern classici più lunghi né la presenza di pattern barrati.

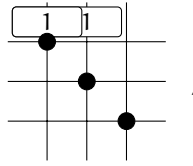
Per quanto quindi questo strumento si sia rivelato molto utile per avere un'idea di quali pattern aspettarsi prima di ricercarli, soprattutto durante le composizioni più complesse che presenteremo in seguito, non può sostituire del tutto un approccio più formale e teorico, come l'applicazione di uno degli algoritmi introdotti prima.



4.2.2 Composizione di queuesort con bubblesort




$$(Q \circ B)^{-1}(Av(21)) = B^{-1}Q^{-1}(Av(21)) = B^{-1}(Av(321))$$

Si osserva che le inversioni in 321 sono (3, 2), (3, 1), (2, 1) e l'unico pattern minimo che le contiene tutte è appunto 321, che quindi è l'unico candidato.

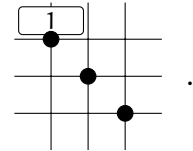
In questo caso a tutte le inversioni deve essere applicata la prima condizione, a parte a (2, 1) perché la condizione è già soddisfatta da 3. Il pattern decorato che si ottiene è:



ovvero l'unione di  e . Tuttavia questi due pattern possono

essere semplificati nel primo: si nota infatti che , che si ottiene dalle condizioni per (3, 2) soddisfa anche le condizioni per (3, 1) e inoltre vale che  \preceq .

Quindi le controimmagini del pattern 321 tramite bubblesort sono date da:



Questo mesh pattern corrisponde ai pattern classici 4321, 3421.

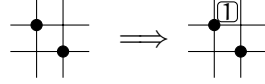
$$Q \circ B^{-1}(Av(21)) = Av(4321, 3421)$$

4.3 COMPOSIZIONI DI UN OPERATORE DI ORDINAMENTO CON STACKSORT

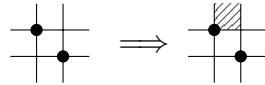
L'algoritmo per le controimmagini di *stacksort* è abbastanza simile a quello per *bubblesort*.

Si ricava una lista di candidati in modo equivalente al primo algoritmo, e ogni candidato viene processato applicando le seguenti regole ad ogni inversione (b, a) presente nel candidato:

se (b, a) è presente anche nell'immagine allora deve essere presente un elemento $c > b$ tra b e a che fa uscire b dalla pila prima che a vi entri



se invece (b, a) non è presente nell'immagine allora si deve escludere la presenza di tale elemento



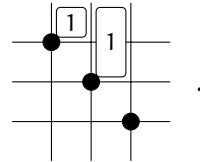
4.3.1 Composizione di *queuesort* con *stacksort*

$$(Q \circ S)^{-1}(Av(21)) = S^{-1}Q^{-1}(Av(21)) = S^{-1}(Av(321))$$

Si ricercano dunque le controimmagini di 321 secondo *stacksort*.

321 è l'unico candidato, quindi sappiamo che tutte le controimmagini devono contenere il pattern 321.

L'unione delle condizioni per le inversioni $(3, 2)$, $(2, 1)$ minimizza anche la condizione per l'inversione $(3, 1)$, dunque il pattern che si ottiene è il seguente:



Per cercare i pattern classici equivalenti si introducono i seguenti elementi:

$3^+ > 3$ rappresenta l'elemento da inserire nella decorazione di sinistra

$2^+ > 2$ rappresenta l'elemento da inserire nella decorazione di destra

Dunque i pattern classici corrispondenti al mesh-pattern trovato devono essere nella forma 33^+22^+1 .

Se $2^+ < 3$ allora la controimmagine assume la forma del pattern 45231.

Altrimenti se $2^+ > 3$, si possono ottenere due diverse controimmagini:

$3^+ > 2^+$ genera il pattern 35241

$2^+ > 3^+$ genera il pattern 34251

$$(Q \circ S)^{-1}(Av(21)) = Av(34251, 35241, 45231)$$

4.3.2 Composizione di *bubblesort* con *stacksort*

$$(B \circ S)^{-1}(Av(21)) = S^{-1}B^{-1}(Av(21)) = S^{-1}(Av(231, 321))$$

Già dall'analisi della composizione precedente è risultato che $S^{-1}(Av(321)) = Av(34251, 35241, 45231)$ quindi è necessario calcolare solo $S^{-1}(Av(231))$. Quest'ultimo risultato è stato ampiamente studiato in letteratura, in quanto analogo al caso di una variante di *stacksort* che utilizza 2 pile. Il risultato che si ottiene è dunque che $S^{-1}(Av(231)) = Av(2341, 3\bar{5}241)[7]$. Si uniscono dunque i due risultati:

$$S^{-1}(Av(231)) = Av(2341, 3\bar{5}241), S^{-1}(Av(321)) = Av(34251, 35241, 45231)$$

Si osserva che $2341 \preceq 34251$, quindi 34251 non è minimo.

Inoltre i pattern $35241, 3\bar{5}241$ insieme, possono essere semplificati dal pattern 3241, dato che questo debba essere evitato sia quando è presente un elemento 5 tra 3 e 2, sia quando è assente. Questo ci permette di semplificare anche 34251, dato che contiene un'occorrenza del pattern 3241 nella sottosequenza 3251

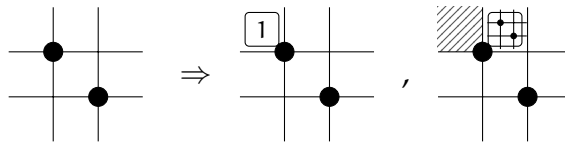
Il risultato che si ottiene è:

$$(B \circ S)^{-1}(Av(21)) = Av(2341, 3241, 45231)$$

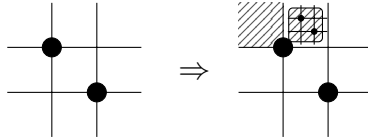
4.4 COMPOSIZIONI DI UN OPERATORE DI ORDINAMENTO CON *queue-sort*

Come per i casi precedenti si stabilisce una lista di pattern candidati, e li si esaminano ad uno ad uno, applicando le seguenti condizioni ad ogni inversione (b, a) :

Se (b, a) è presente anche nell'immagine è perché non viene ordinata da una passata di *queuesort*, questo si verifica sicuramente quando b non è un LTR massimo (a non lo è mai, in quanto minore di b), se invece b risulta essere un LTR massimo allora deve essere presente un'ulteriore inversione (d, c) tra b e a , così che d si accodi e c provochi l'estrazione di b prima che a possa effettuare un bypass. Queste condizioni si applicano decorando l'inversione in uno dei modi seguenti



Se un'inversione contenuta nel candidato non è presente nell'immagine significa che nessuna delle condizioni espresse prima viene soddisfatta, la decorazione che si applica rappresenta una negazione delle condizioni mostrate prima

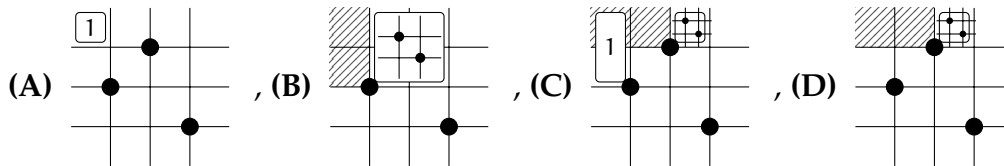


4.4.1 Composizione di stacksort con queuesort

$$S \circ Q^{-1}(Av(21)) = Q^{-1}S^{-1}(Av(21)) = Q^{-1}(Av(231))$$

Si ricercano dunque le controimmagini di 231 secondo queuesort. Gli stessi elementi che formano un pattern 231 nell'immagine possono formare nella controimmagine lo stesso pattern 231 o un pattern 321. Si esaminano i due casi separatamente.

Si applicano alle inversioni di 231 le possibili condizioni espresse precedentemente, ottenendo i seguenti mesh-pattern:



Si osserva che i pattern (D) e (C) contengono un'occorrenza del pattern (B) e dunque evitare (B) comprende evitare anche (C) e (D).

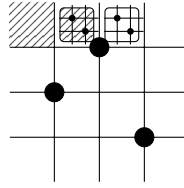
Il pattern (A) corrisponde al pattern classico 4231, mentre il pattern (B) è simile ad un pattern 2431 con alcune aree oscure.

Esaminiamo questo caso in cui una permutazione eviti il pattern (B) più approfonditamente: una permutazione evita questo pattern se in ogni occorrenza di un pattern 2431 l'elemento corrispondente al 2 del pattern non è un LTR massimo. Si prenda il primo LTR massimo alla sinistra dell'elemento 2 (tale elemento esiste dato che stiamo esaminando il caso di una permutazione che eviti (B), incluse le aree oscure): questo entra in coda, 2 viene aggiunto all'output, e almeno uno tra l'elemento precedente e gli elementi della coppia 43 deve essere aggiunto all'output; ognuno di questi elementi è maggiore di 2 e dunque si va a formare un pattern 231 quando 1 bypassa. Se ne deduce che, ai fini di evitare la

produzione di pattern 231, il fatto che una permutazione eviti il pattern (B) implica che essa eviti anche il pattern 2431, dato che la presenza di elementi nelle aree oscure non previene la formazione di pattern 231.

Si cercano adesso le controimmagini di 231 che contengono 321, applicando le regole dell'algoritmo alle inversioni di 321.

Applicando le regole esposte, ed escludendo quelle che si contraddicono, si ottiene il seguente mesh-pattern:



che contiene un'occorrenza del pattern 2431 trovato prima, quindi anche questo può essere semplificato.

Si ottiene che:

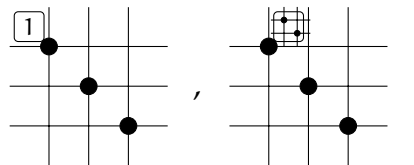
$$S \circ Q^{-1}(Av(21)) = Av(2431, 4231)$$

4.4.2 Composizione di *bubblesort* con *queuesort*

$$B \circ Q^{-1}(Av(21)) = Q^{-1}(B^{-1}(Av(21))) = Q^{-1}(Av(231, 321))$$

L'insieme $Q^{-1}Av(231)$ è già stato calcolato nella sezione precedente, quindi resta da cercare l'insieme $B^{-1}Av(321)$ ed eseguire l'intersezione tra i due.

321 è l'unico candidato per la ricerca delle controimmagini, ed inoltre l'elemento 3 soddisfa già una delle condizioni per l'inversione (2, 1). Applicando le condizioni alle inversioni (3, 1), (3, 2), si ottengono i seguenti mesh-pattern:



Entrambi corrispondono a pattern classici, rispettivamente 4321, 35421. Si noti che il secondo contiene un'occorrenza del primo, che quindi è l'unico da considerare. Si unisce questo risultato a quelli del capitolo precedente.

$$B \circ Q^{-1}(Av(21)) = Av(2431, 4231, 4321)$$

4.5 CONTENITORI POP

Lo studio di composizioni di operatori che utilizzano contenitori POP si rivela piú difficile, in quando si tratta di casi non approfonditi in letteratura.

Sono tuttavia noti i pattern che rendono una permutazione non ordinabile, come presentato nel capitolo 2, ovvero:

$$S_{POP}^{-1}(Av(21)) = Av(231, 312)$$

$$Q_{POP}^{-1}(Av(21)) = Av(321, 2413)$$

Grazie a questi dati, è possibile individuare un caso di facile risoluzione: ovvero quello in cui un algoritmo che usa contenitori POP è concatenato ad un operatore classico.

$$(X_{POP} \circ Y)(\pi) = X_{POP}(Y(\pi)) \Rightarrow (X_{POP} \circ Y)^{-1}(\pi) = Y^{-1}X_{POP}^{-1}(\pi) = Y^{-1}(Av(M))$$

Conoscendo l'insieme di pattern M che rendono la permutazione non ordinabile dall'operatore POP, basterá cercare le loro controimmagini secondo l'operatore classico con i metodi usati finora, come verrà mostrato nelle sezioni future.

4.5.1 POP-Queuesort - dettagli ulteriori

L'algoritmo POP-queuesort richiede una diversa analisi rispetto agli altri algoritmi.

Esistono diverse versioni di *POP queuesort* ed in particolare ne esistono due ottimali[4]: *Min* e *Cons*.

min in questa versione l'operazione di POP viene eseguita se il primo elemento della coda è il piú piccolo elemento non ancora in output; se l'elemento in input è maggiore dell'ultimo elemento della coda (o se la coda è vuota) allora viene accodato mentre negli altri casi, se l'elemento dell'input è minore della testa della coda allora bypassa altrimenti si esegue un POP.

cons questa versione si basa sull'idea di avere sempre elementi consecutivi nella coda; è la versione che verrà adottata in questa tesi; da qui in avanti per lavorare con una coda POP utilizzeremo esclusivamente l'algoritmo *Cons*[4].

Nell'algoritmo *Cons* si definisce una coda POP come un insieme Q , inizialmente vuoto, di interi su cui è possibile eseguire le seguenti operazioni:

ENQUEUE(π_i , Q): inserisce l' i -esimo elemento della permutazione π in Q , accodandolo alla coda;

POP(Q): estrae gli elementi contenuti in Q e li aggiunge all'output nello stesso ordine a cui sono stati aggiunti a Q ;

BYPASS(π_i): posiziona l' i -esimo elemento di π nell'output.

Inoltre si utilizzano le seguenti notazioni:

BACK(Q): indica il valore dell'ultimo elemento aggiunto a Q ,

FRONT(Q): restituisce il valore dell'elemento che sta in cima a Q .

Algorithm 4 *Cons* - POP Queuesort

```

1:  $Q \leftarrow \emptyset$ 
2: for  $i = 1$  to  $n$  do
3:   if  $Q = \emptyset$  or  $\pi_i = \text{Back}(Q) + 1$  then
4:     Enqueue( $\pi_i$ )
5:   else
6:     if  $\text{Front}(Q) > \pi_i$  then
7:       Bypass( $\pi_i$ )
8:     else
9:       Pop( $Q$ )
10:      Enqueue( $\pi_i$ )
11:    end if
12:  end if
13: end for
14: if  $Q \neq \emptyset$  then
15:   Pop( $Q$ )
16: end if

```

La dimostrazione che *Cons* (così come *Min*) sia un algoritmo ottimale nella classe degli algoritmi *POP-queuesort* segue dal fatto che esso ordina tutte e sole le sequenze dell'insieme $\text{Av}(321, 2413)$, che sono esattamente tutte e sole le permutazioni ordinabili con una coda POP[4].

4.6 COMPOSIZIONI CHE INCLUDONO OPERATORI POP

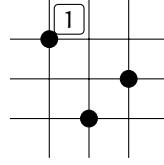
4.6.1 *Composizione di POP-stacksort con stacksort*

$$(S_{\text{POP}} \circ S)^{-1}(\text{Av}(21)) = S^{-1}(\text{Av}(231, 312))$$

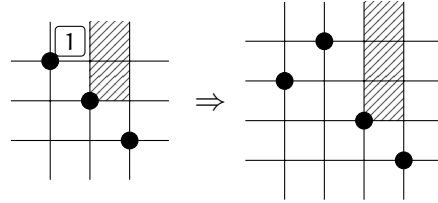
È già noto dalle composizioni precedenti che $S^{-1}(\text{Av}(231)) = \text{Av}(2341, 3\bar{5}241)$, quindi restano da calcolare le controimmagini di 312 tramite S .

I pattern candidati che possono generare 312 sono quelli che contengono le coppie $(3, 1)$, $(3, 2)$, ovvero 321, 312.

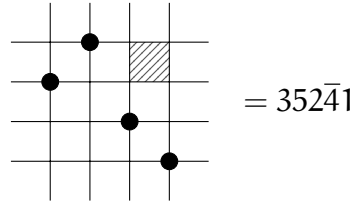
Si considera il candidato 312. Le sue condizioni che l'algoritmo impone di applicare alle sue inversioni sono soddisfatte dal seguente mesh-pattern, che corrisponde al pattern classico 3412.



Si considera il candidato 321. Le sue inversioni sono $(3, 2)$, $(3, 1)$, $(2, 1)$. Si noti che l'inversione $(2, 1)$ deve essere ordinata. Applicando tutte le condizioni si ottiene il seguente mesh-pattern:



Si esaminano le aree oscurate: se un elemento fosse presente nell'area oscurata più in basso questo formerebbe un'occorrenza del pattern 3412 con gli elementi precedenti, dunque quella specifica area oscurata, ai fini delle controimmagini e tenendo conto dei pattern precedenti, non fornisce nuove informazioni e dunque può essere trascurata al fine di semplificare il pattern. Con una motivazione analoga si può escludere anche l'area oscurata in alto, dato che la presenza di un elemento in tale area forma un pattern 2341. Dalle semplificazioni si ottiene il seguente mesh-pattern:



$$(S_{\text{POP}} \circ S)^{-1}(Av(21)) = Av(2341, 3412, 3421, 3\bar{5}241, 352\bar{4}1)$$

4.6.2 Composizione di POP-stacksort con bubblesort

$$(S_{\text{POP}} \circ B)^{-1}(Av(21)) = B^{-1}(Av(231, 312))$$

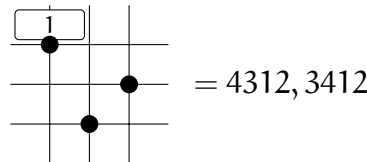
La controimmagine tramite B di 231 è stata calcolata: $B^{-1}(Av(231)) = (S \circ B)^{-1}(Av(21)) = Av(2341, 2431, 3241, 4231)$.

Si cercano dunque le controimmagini di 312 secondo bubblesort.

Gli elementi del pattern 312 nell'immagine possono solo formare un pattern 312 o 321 nella controimmagine.

Si può osservare come gli stessi elementi di un pattern 321 non possono divenire un 312 . La presenza dell'elemento maggiore del pattern all'inizio di essa evita l'ordinamento degli elementi successivi.

Perché un pattern 312 rimanga invariato è sufficiente la presenza di un elemento maggiore di tutto il pattern in posizione tale da evitare l'ordinamento di 3 con qualsiasi elemento, applicando le regole descritte prima per *bubblesort*:



$$(S_{\text{POP}} \circ B)^{-1}(Av(21)) = Av(2341, 2431, 3241, 3412, 4231, 4312)$$

4.6.3 Composizione di POP-stacksort con queuesort

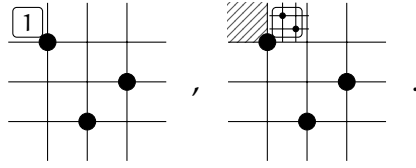
$$(S_{\text{POP}} \circ Q)^{-1}(Av(21)) = Q^{-1}(Av(231, 312))$$

È già noto dai calcoli nelle sezioni precedenti che $Q^{-1}(Av(231)) = Av(4231, 2431)$.

Si cercano le controimmagini di 312 tramite queuesort.

I candidati per la produzione di controimmagini di 312 sono $312, 321$.

Applicando le regole dell'algoritmo per le controimmagini secondo queuesort alle inversioni $(3, 1), (3, 2)$ si ottengono i seguenti mesh-pattern:



Il primo dei due mesh pattern corrisponde al pattern classico 4312, che è contenuto anche nel secondo.

Usando invece 321 come candidato si osserva come l'inversione (2,1) richieda di essere invertita, ma questo non può verificarsi dato che la presenza dell'elemento 3 rende per 2 impossibile essere un LTR massimo. Dunque il pattern stesso contraddice le condizioni dell'algoritmo e non è possibile trovare una controimmagine partendo da questo candidato.

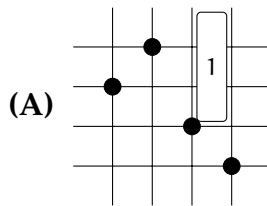
$$(S_{\text{POP}} \circ Q)^{-1}(\text{Av}(21)) = \text{Av}(2431, 4231, 4312)$$

4.6.4 Composizione di POP-queuesort con stacksort

$$Q_{\text{POP}} \circ S^{-1}(\text{Av}(21)) = S^{-1}Q_{\text{POP}}^{-1}(\text{Av}(21)) = S^{-1}(\text{Av}(321, 2413))$$

Si osserva la controimmagine di 321 tramite S è già stata calcolata: $S^{-1}(\text{Av}(321)) = (Q \circ S)^{-1}(\text{Av}(21)) = \text{Av}(34251, 35241, 45231)$.

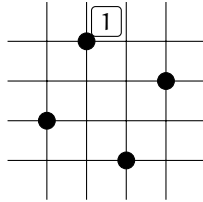
Si osserva come questi pattern hanno una forma in comune: essi sono infatti tutti composti da una non inversione seguiti da un pattern 231, dove gli elementi 2, 1 di quest'ultimo sono minori della non inversione iniziale. Di seguito si illustra questa forma con un pattern decorato che chiamiamo (A).



In questo caso risulta utile evidenziare la forma di questo specifico pattern in quanto ricorrente anche in quelli che troveremo successivamente, e quindi renderà più semplice individuare le occorrenze di questi pattern già noti in quelli che verranno trovati per semplificarli.

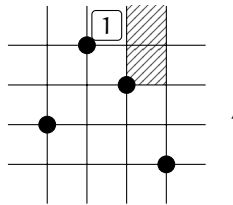
Si ricerca le controimmagini del pattern 2413 secondo stacksort. I candidati sono: 2413, 2431, 4213, 4231, 4321.

2413: Le inversioni di 2413 sono $(2, 1)$, $(4, 1)$, $(4, 3)$. Si osserva che 4 soddisfa già la condizione per l'inversione $(2, 1)$, e decorando il pattern per soddisfare la condizione per $(4, 1)$ si ottiene



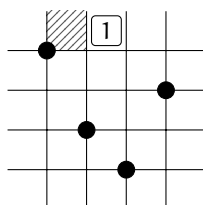
che soddisfa anche la condizione per l'inversione $(4, 3)$, il mesh-pattern così ottenuto equivale al pattern classico 24513, che è l'unica controimmagine che si possa ricavare da questo candidato.

2431: Le inversioni di 2431 sono $(2, 1)$, $(4, 3)$, $(4, 1)$, $(3, 1)$. La condizione per $(2, 1)$ è già soddisfatta, applicando le ulteriori condizioni si ottiene il seguente mesh-pattern



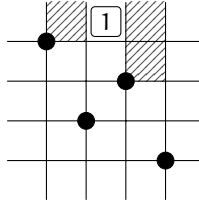
che equivale al pattern classico 24531 con alcune aree barrate. Nell'esaminare le aree barrate si può osservare che se un elemento le occupasse questo formerebbe un'occorrenza del pattern (A) con gli altri elementi del pattern a parte il 2. Se ne deduce che le aree barrate ai fini delle controimmagini e tenendo conto dei pattern precedenti, non forniscono nuove informazioni e dunque possono essere trascurate al fine di semplificare il pattern, ottenendo il pattern classico 24531. Simili osservazioni verranno utilizzate anche per i successivi candidati.

4213: Le inversioni di 4213 sono $(4, 2)$, $(4, 1)$, $(4, 3)$, $(2, 1)$. Di queste 4, 2 non è presente in 2413, e quindi provoca l'inserimento di un'area oscurata. Per soddisfare la condizione relativa all'inversione $(4, 1)$ si procede ad aggiungere una decorazione che risulta soddisfare anche le condizioni per le inversioni $(4, 3)$ e $(2, 1)$.



L'area oscurata può essere trascurata dato che la sua occupazione provoca un'occorrenza del pattern (A), e dunque si ottiene il pattern classico 42513.

4231: Le inversioni di 4231 sono (4, 2), (4, 3), (4, 1), (2, 1), (3, 1). Le condizioni delle inversioni (4, 2) e (3, 1) richiedono di inserire delle aree oscurate mentre la decorazione aggiunta per l'inversione (4, 3) soddisfa anche la condizione per (4, 1), (2, 1).



Si osserva come le aree oscurate possano essere trascurate dato che la loro occupazione causa un'occorrenza del pattern (A), semplificando come nei casi precedenti si ottiene il pattern 42531.

4321: Le inversioni del pattern 4321 sono (4, 3), (4, 2), (4, 1), (3, 2), (3, 1), (2, 1). Applicando tutte le regole si nota come queste siano contraddittorie tra di loro, dunque questo candidato non può generare controimmagini.

Si uniscono tutti i risultati ottenuti:

$$Q_{POP} \circ S^{-1}(Av(21)) = Av(24513, 24531, 34251, 35241, 42513, 42531, 45231)$$

4.6.5 Composizione di POP-queuesort con bubblesort

$$(Q_{POP} \circ B)^{-1}(Av(21)) = B^{-1}Q_{POP}^{-1}(Av(21)) = B^{-1}(Av(321, 2413))$$

La classe di pattern $B^{-1}(Av(321))$ è già stata calcolata:

$$B^{-1}(Av(321)) = (Q \circ B)^{-1}(Av(21)) = Av(3421, 4321)$$

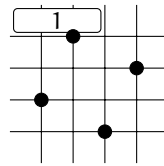
Si ricerca adesso la classe di pattern corrispondente a $B^{-1}(Av(2413))$. La lista dei candidati è la seguente: 2413, 2431, 4213, 4231, 4321.

OSSERVAZIONE: i pattern 2431, 4231, 4321 contengono l'inversione (3, 1) che non è presente nel pattern dell'immagine. Questa specifica inversione è in posizione tale che nessuno dei due elementi sia un LTR massimo, e dunque l'inversione non può essere ordinata. In questi 3 pattern infatti, l'elemento 4 si trova all'interno dell'area oscurata che si

dovrebbe aggiungere durante l'applicazione dell'algoritmo per le controimmagini di bubblesort all'inversione $(3, 1)$. Ne consegue che i pattern 2431, 4231, 4321, pur essendo nella lista di candidati non possono generare controimmagini del pattern 2413.

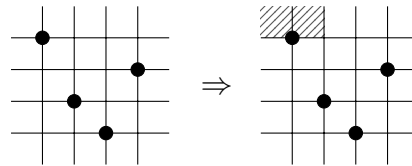
Si ricercano ora le controimmagini relative agli altri candidati:

2413: Si osserva che 4 soddisfa già la condizione per l'inversione $(2, 1)$, dunque si prende l'intersezione tra le condizioni per le inversioni $(4, 3)$, $(4, 1)$ ottenendo il seguente pattern decorato:

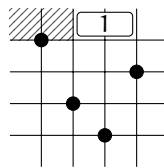


Da questo si ottengono i pattern classici 52413, 25413, 24513.

4213: L'inversione $(4, 2)$ deve essere ordinata, in quanto non presente in 2413.



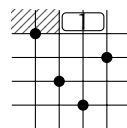
Si osserva poi che 4 soddisfa già le condizioni per l'inversione $(2, 1)$, mentre applicando le condizioni per le altre inversioni, tenendo conto dell'area oscurata si ottiene:



Trovare l'insieme di pattern classici equivalente a quest'ultimo mesh-pattern non è immediato come lo è in altri casi, dunque si continuerà ad adottare la notazione dei mesh-pattern, invece di quella dei pattern classici, per rappresentare questa soluzione.

Si uniscono i risultati ottenuti.

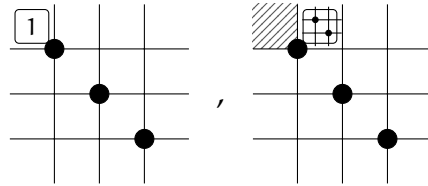
$$(Q_{\text{POP}} \circ B)^{-1}(\text{Av}(21)) = \text{Av}(3421, 4321, 24513, 25413, 52413,$$



4.6.6 Composizione di POP-queuesort con queuesort

$$Q_{\text{POP}} \circ Q^{-1} \text{Av}(21) = Q^{-1} Q_{\text{POP}}^{-1} (\text{Av}(21)) = Q^{-1} (\text{Av}(321, 2413))$$

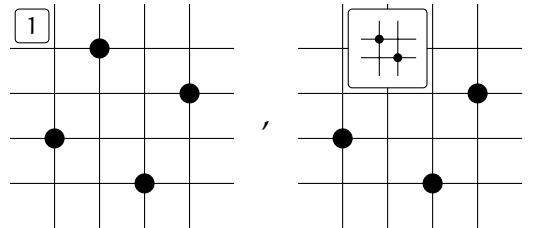
Si ricercano le controimmagini di 321. Si osserva che 3 soddisfa già una delle condizione per l'inversione (2, 1). Applicando le condizioni alle altre inversioni, i pattern minimi che si ottengono sono i seguenti:



Il primo corrisponde al pattern classico 4321 che, essendo contenuto anche nel secondo, risulta essere l'unica controimmagine individuata da questo candidato.

Si ricercano adesso le controimmagini di 2413. Le inversioni di 2413 sono (2, 1), (4, 1), (4, 3), che ammettono i candidati 2413, 2431, 4213, 4231, 4321. Di questi 4321 è già stato trovato prima, dunque non vi verrà applicato l'algoritmo dato che ogni sua controimmagine ne contiene sicuramente un'occorrenza.

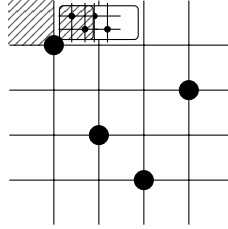
2413: I pattern minimi che si ottengono applicando le condizioni sono i seguenti:



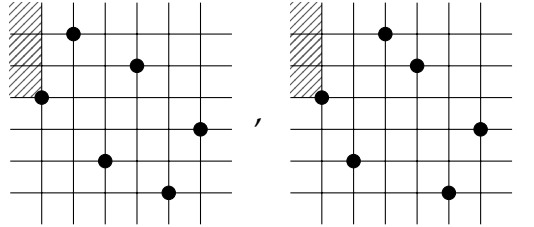
Le condizioni relative a tutte le inversioni del candidato producono più di due pattern, ma per ragioni di brevità si mostrano solo questi, di cui tutti gli altri contengono occorrenze, omettendo le semplificazioni. I due mesh-pattern generano rispettivamente i pattern classici 52413 e 25413.

2431: In questo candidato si osserva come l'inversione (4, 3) contraddica la condizione per l'inversione (2, 1), inoltre l'elemento 4 contraddice la condizione dell'inversione (3, 1), dunque questo candidato non può produrre controimmagini.

4213: Applicando le condizioni si ottiene il seguente mesh-pattern



Le condizioni richiedono la presenza di un'inversione di elementi maggiori di 4 tra 4 e 1 e l'assenza di un'inversione simile tra 4 e 2. Questo può generare i seguenti mesh-pattern:



Si osserva come il primo dei due contenga un'occorrenza del pattern 52413 e il secondo di 25413. Dunque le controimmagini ottenute da questo candidato possono essere semplificate.

4231: Anche in questo caso, similmente al caso precedente è richiesta la presenza di un'inversione di elementi maggiori di 4 tra 4 e 3 e l'assenza di un'inversione simile tra 4 e 2. Si noti tuttavia come questa eventualità produca necessariamente un pattern 4321, in particolare considerando l'inversione composta da elementi superiori a 4 e gli elementi 3 e 1. Dunque il fatto che una permutazione eviti un pattern 4321 comprende anche che essa eviti il pattern trovato e quindi questo candidato non produce controimmagini utili.

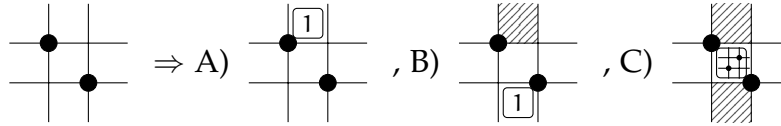
$$Q_{\text{POP}} \circ Q^{-1} \text{Av}(21) = \text{Av}(4321, 25413, 52413)$$

4.6.7 Presentazione dell'algoritmo per la ricerca di controimmagini secondo l'operatore POPstacksort

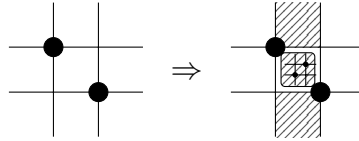
Si ricercano le condizioni che un candidato deve soddisfare per produrre una voluta immagine. Come negli altri casi, si esaminano le inversioni nel candidato, per ogni inversione (b, a) che non viene ordinata da una passata di POPstacksort si verifica una delle seguenti condizioni:

- A) un elemento $c > b$ è presente tra b e a , questo provoca il POP prima che l'algoritmo arrivi ad esaminare a ;
- B) un elemento $a^- < a$ (e nessun elemento $c > b$) è presente tra b e a , quindi a^- viene inserito nella pila ed è a a provocare il POP;
- C) tra i due elementi ma non c'è nessun elemento minore di a o maggiore b ma c'è una coppia ordinata, ovvero esistono $b^-, a^+ : a < a^+ < b^- < b$ e $b \dots a^+ \dots b^- \dots a$ è una sottosequenza nel candidato.

Quindi per trovare una controimmagine secondo POPstacksort occorre applicare una tra le seguenti decorazioni, per ogni inversione che debba restare tale, al mesh-pattern del candidato:



Se invece ci si vuole assicurare che un'inversione venga ordinata, la decorazione da applicare consiste semplicemente nella "negazione" di tutte quelle espresse finora:



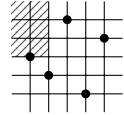
NOTA: L'algoritmo, che è stato presentato per motivi di completezza, permette di ricavare le controimmagini delle seguenti composizioni: $S_{POP} \circ S$, $S_{POP} \circ Q$, $S_{POP} \circ B$. Questi casi non sono verranno esaminati dato che la loro trattazione richiede troppi dettagli per essere inclusa tra gli argomenti di questa tesi.

4.6.8 Osservazione sulle analisi di composizioni di operatori di ordinamento con POP-queuesort

Non si ha a disposizione un algoritmo per la ricerca di controimmagini di pattern tramite *Cons*. Per questo motivo le classi di pattern delle combinazioni mostrate di seguito, ovvero quelle in cui si compone POP-queuesort ad un altro algoritmo di ordinamento, verranno ricercate con un metodo diverso da quello adottato finora: si presenteranno delle congetture su quali classi di pattern che si ritengono possano coincidere con le controimmagini cercate e si verificherà che l'insieme di permutazioni ordinabili dalla composizione e l'insieme mostrato nella congettura coincidano.

4.6.9 Composizione di *stacksort* con *POP-queuesort*

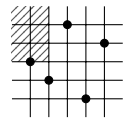
Grazie a vari risultati ottenuti applicando i software mostrati nel capitolo 3, è stato possibile formulare la seguente congettura:

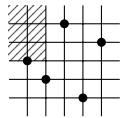
$$S \circ Q_{POP}^{-1}Av(21) = Av(2431, 4231, 23514, \text{ )$$

Questo equivale a cercare le controimmagini del pattern 231 secondo *Cons*, dato che:

$$S \circ Q_{POP}^{-1}Av(21) = Q_{POP}^{-1}(S^{-1}(Av(21))) = Q_{POP}^{-1}(Av(231))$$

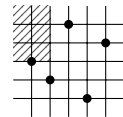
Per dimostrare la congettura occorre dimostrare la doppia inclusione:

$$(A) \text{ Cons}^{-1}Av(231) \subseteq Av(2431, 4231, 23514, \text{ )$$

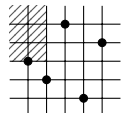
$$(B) Av(2431, 4231, 23514, \text{ ) \subseteq \text{Cons}^{-1}Av(231)$$

NOTA: Per tutta la dimostrazione i valori che formano i pattern saranno indicati da a, b, c, d ed e (quando presente), n indicherà la lunghezza della permutazione π e si intende $1 \leq a < b < c < d < e \leq n$.

OSSERVAZIONE: Sia π una permutazione in input all'algoritmo *Cons*, tutti gli elementi di π che non sono LTR massimi effettuano un bypass. Un LTR massimo può accodarsi se e solo se è il consecutivo dell'elemento in fondo alla coda e provoca un *pop* in tutti gli altri casi.

$$(A) \text{ Cons}^{-1}Av(231) \subseteq Av(2431, 4231, 23514, \text{ )$$

Dimostrazione. Dobbiamo dimostrare che tutte le permutazioni la cui im-

mage evita il pattern 231 evitano i pattern 2431, 4231, 23514, , ovvero che l'immagine di ogni permutazione che contiene almeno uno di questi pattern contiene un pattern 231.

2431

$$\pi = \dots b \dots d \dots c \dots a \dots$$

Le non inversioni (b, d) e (b, c) saranno presenti anche nell'immagine e inoltre c, a , non essendo LTR massimi, effettuano un bypass. Si osserva che se b effettua un bypass si ha necessariamente un pattern 231 nella sottosequenza bca . Se b entra in coda allora d provoca necessariamente un pop (non può essere accodato alla stessa coda che contiene b , perché la coda è consecutiva e c , che è un valore intermedio tra b e d , è più avanti nell'input) e l'immagine contiene la sottosequenza bca .

4231

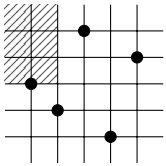
$$\pi = \dots d \dots b \dots c \dots a \dots$$

Nessuno degli elementi che formano la sottosequenza bca contenuta in π è un LTR massimo, quindi bypassano tutti e la sequenza è contenuta anche in $\text{Cons}(\pi)$.

23514

$$\pi = \dots b \dots c \dots e \dots a \dots d \dots$$

Le non inversioni sono presenti anche nell'immagine, in particolare la non inversione (b, c) , inoltre a non è un LTR massimo quindi effettua un bypass. Si vuole mostrare che la non inversione (b, c) viene sempre inserita nell'output prima di a . Se b, c effettuano un bypass si ha sicuramente bca nell'immagine. Se entrambi entrano in coda almeno un pop viene provocato da e prima che si arrivi ad a . Anche se b effettua un bypass e c entra in coda quest'ultimo viene sicuramente aggiunto all'output prima di a , eventualmente da e .



Una permutazione che contiene occorrenza di questo pattern ha la forma:

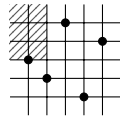
$$\pi = \dots c \dots b \dots e \dots a \dots d \dots$$

dove nessun valore precedente a b è maggiore di c .

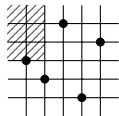
Si osserva che b, a effettuano un bypass. c è un LTR massimo, quindi sia che provochi un pop o meno, verrà inserito in coda, dopodiché si

verifica il bypass di b , e c viene sicuramente estratto dalla coda prima di arrivare ad a (eventualmente è e a provocare il pop). Si può escludere che c esca dalla coda prima che b effettui un bypass perché prima di b nessun elemento è maggiore di c .

Si forma così la sottosequenza bca . □

$$(B) \text{Av}(2431, 4231, 23514, \text{ ) \subseteq \text{Cons}^{-1}\text{Av}(231)$$

Dimostrazione. Dobbiamo dimostrare che tutte le permutazioni che evita-

no i pattern $2431, 4231, 23514$,  sono contenute anche nell'insie-

me $\text{Cons}^{-1}\text{Av}(231)$, ovvero si dimostrerà che ogni permutazione π tale che $231 \preceq \text{Cons}(\pi)$ contiene uno dei pattern elencati.


Si osserva che se 3 elementi nell'immagine $\text{Cons}(\pi)$ formano un pattern 231 , quegli stessi elementi devono formare un pattern 231 o 321 nella controimmagine, dato che Cons non produce nuove inversioni. Si analizzano questi due casi separatamente.

$231 \preceq \pi$:

$$\pi = \dots b \dots c \dots a \dots$$

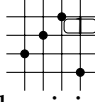
$$\text{Cons}(\pi) = \dots b \dots c \dots a \dots$$

La non inversione (b, c) resta anche nell'immagine. Si vuole definire l'ordine relativo degli altri elementi di π rispetto a a, b, c perché le inversioni $(b, a), (c, a)$ non vengano rimosse da Cons .

Se la forma di π è tale che b entri in coda, per mantenere la sottosequenza bca è necessario che c non sia un LTR massimo. Questa condizione è soddisfatta se π contiene un'occorrenza del mesh-pattern , ovvero se contiene uno dei pattern classici 4231 o 2431 .

Se la forma di π è invece tale per cui c entri in coda, devono essere presenti degli elementi in π che provocano un pop prima che a possa bypassare (si noti come queste considerazioni si applicano sia nel caso in cui b sia in coda con c al momento del pop, sia in quelli in cui b è già stato inserito nell'output in seguito ad un bypass). Per assicurare il pop è sono necessari due nuovi elementi: e, d dove e deve essere posizionato

tra c e a mentre d dopo di e . L'elemento e assicura il pop se è maggiore della testa della coda (se è minore tutti gli elementi del pattern effettuano un bypass) ma non può accodarsi, in quanto d non può ancora essere in coda e quindi e non può essere consecutivo alla coda. Queste condizioni


sono soddisfatte dal mesh-pattern , quindi si deve avere che π contenga almeno uno tra i pattern classici $23541, 23514$. Il primo dei due pattern classici viene ignorato, in quanto contiene uno dei pattern già trovati prima: $2431 \preceq 23541$.

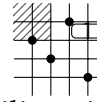
$321 \preceq \pi$:

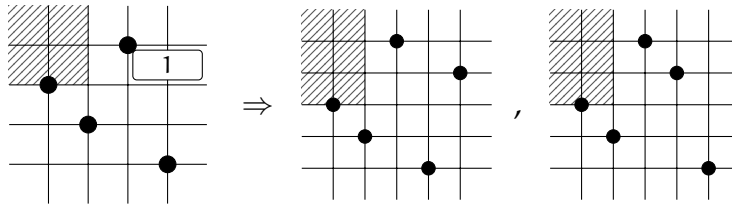
$$\pi = \dots c \dots b \dots a \dots$$

$$\text{Cons}(\pi) = \dots b \dots c \dots a \dots$$

Ci chiediamo quale deve essere l'ordine relativo degli altri elementi di π rispetto a a, b, c per ottenere che l'inversione $(3, 2)$ venga ordinata mentre le altre restino invariate.

Per ordinare (b, c) , c deve essere un LTR massimo, in modo che possa entrare nella coda, e non deve verificarsi nessun pop tra c e b . Queste condizioni sono rappresentate dalle aree oscure nel mesh pattern .

Per forzare un pop tra c e a è necessario introdurre due elementi d , e come nel caso precedente: si ottiene così il mesh-pattern , la cui parte decorata permette rappresenta l'unione delle possibili posizioni del valore d nei due successivi mesh-pattern mostrati:



Degli ultimi due mesh pattern mostrati, si ignora il secondo in quanto contiene un'occorrenza del pattern 2431 , già trovato in precedenza, nella sottosequenza $beda$. \square

4.6.10 Composizione di queuesort con POP-queuesort

Similmente a come fatto per la sezione precedente, si vuole adesso dimostrare il seguente risultato:

$$Q \circ Q_{\text{POP}}^{-1} \text{Av}(21) = \text{Av}(4321, 35214, 35241)$$

Questo equivale a cercare le controimmagini del pattern 321 secondo *Cons*, dato che:

$$Q \circ Q_{\text{POP}}^{-1} \text{Av}(21) = Q_{\text{POP}}^{-1}(Q^{-1}(\text{Av}(21))) = Q_{\text{POP}}^{-1}(\text{Av}(321))$$

Per dimostrare la congettura occorre dimostrare la doppia inclusione:

$$(A) \text{ Cons}^{-1} \text{Av}(321) \subseteq \text{Av}(4321, 35214, 35241)$$

$$(B) \text{ Av}(4321, 35214, 35241) \subseteq \text{Cons}^{-1} \text{Av}(321)$$

NOTA: Si adotta la stessa dicitura, per gli elementi che formano i pattern, adottata nella precedente dimostrazione

$$(A) \text{ Cons}^{-1} \text{Av}(321) \subseteq \text{Av}(4321, 35214, 35241)$$

Dimostrazione. Dobbiamo dimostrare che tutte le permutazioni la cui immagine evita il pattern 321 evitano i pattern 4321, 35214, 35241.

Si procede a dimostrare che l'immagine di ogni permutazione che contiene almeno uno di questi pattern contiene un pattern 321.

4321:

$$\pi = \dots d \dots c \dots b \dots a \dots$$

Nessun memebro della sottosequenza *cba* è un LTR massimo, quindi effettuano tutti un bypass e vengono inseriti nell'output nello stesso ordine, formando un pattern 321.

35214:

$$\pi = \dots c \dots e \dots b \dots a \dots d \dots$$

Gli elementi *b*, *a* non sono LTR massimi e quindi effettuano un bypass. Si vuole dimostrare che l'elemento *c* viene sempre inserito nell'output prima dell'inversione (*b*, *a*). Questo è evidente quando *c* effettua un bypass. Se invece *c* entra in coda, anche se in seguito ad un pop, la testa della coda è sicuramente minore di *e*, che quindi deve necessariamente causare un pop (non può accodarsi dato che *c* è in coda e *d* è ancora nell'input), ponendo *c* nell'output prima di *b*.

35241: In questo caso valgono le stesse osservazioni del caso precedente. \square

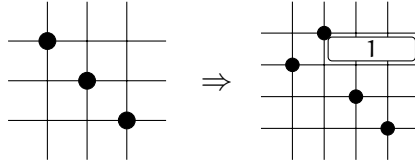
$$(A) \text{Av}(4321, 35214, 35241) \subseteq \text{Cons}^{-1} \text{Av}(321)$$

Dimostrazione. Dobbiamo dimostrare che se una permutazione evita i pattern 4321, 35214, 35241, la sua immagine evita il pattern 321. Si procede a dimostrare che per avere un pattern 321 nell'immagine $\text{Cons}(\pi)$ è necessario che π contenga almeno uno tra i pattern 4321, 35214, 35241.

Poiché *Cons* non produce nuove inversioni gli elementi che c, b, a che formano il pattern 321 nell'immagine devono essere nello stesso ordine relativo anche nella controimmagine.

Se nessuno degli elementi della sottosequenza cba è un LTR massimo allora tutti e 3 gli elementi effettuano un bypass e il pattern 321 è preservato. Questo si verifica quando cba sono i 3 elementi minori di un'occorrenza di un pattern 4321.

Se c invece entra in coda, anche se in seguito ad un pop, deve avvenire un pop prima che b possa effettuare un bypass. Questo si verifica quando un elemento maggiore di e è posto tra c e b ed un ulteriore elemento d è successivo ad e (come mostrato anche nel caso precedente).



Una permutazione contiene un'occorrenza del pattern decorato così ottenuto se contiene uno dei seguenti pattern classici: 35421, 35241, 35214. Di questi il primo viene escluso perché contiene a sua volta un'occorrenza del pattern 4321, già trovato prima. \square

4.6.11 Composizione di bubblesort con POP-queuesort

$$(B \circ Q_{\text{POP}})^{-1}(\text{Av}(21)) = Q_{\text{POP}}^{-1}(B^{-1}(\text{Av}(21))) = Q_{\text{POP}}^{-1}(\text{Av}(231, 321))$$

La classe di pattern ordinabile dalla composizione di bubblesort con *Cons* è data dall'unione dei risultati ottenuti per *stacksort* e *queuesort*.

$$(B \circ Q_{\text{POP}})^{-1}(Av(21)) = Av(2431, 4231, 4321, 23514, 35214, 35241, \text{ } \begin{array}{|c|c|c|c|} \hline \text{shaded} & \bullet & & \\ \hline \text{shaded} & & & \bullet \\ \hline \text{shaded} & \bullet & & \\ \hline \text{shaded} & & \bullet & \\ \hline \end{array})$$

Si osserva che l'insieme non è minimo, in quanto alcuni pattern contengono istanze di altri: $2413 \preceq 35214$, $4231 \preceq 35241$. Eliminati i pattern superflui, si ottiene:

$$(B \circ Q_{\text{POP}})^{-1}(Av(21)) = Av(2431, 4231, 4321, 23514, \text{ } \begin{array}{|c|c|c|c|} \hline \text{shaded} & \bullet & & \\ \hline \text{shaded} & & & \bullet \\ \hline \text{shaded} & \bullet & & \\ \hline \text{shaded} & & \bullet & \\ \hline \end{array})$$

CONCLUSIONI

Si riassume nella seguente tabella i risultati raggiunti durante la trattazione di questa tesi.

Composizione	Permutazioni ordinabili con una sola passata
$S \circ B$	$Av(2341, 2431, 3241, 4231)$
$Q \circ B$	$Av(3421, 4321)$
$Q \circ S$	$Av(34251, 35241, 45231)$
$B \circ S$	$Av(2341, 3241, 45231)$
$S \circ Q$	$Av(2431, 4231)$
$B \circ Q$	$Av(2431, 4231, 4321)$
$S_{POP} \circ S$	$Av(2341, 3412, 3421, 3\bar{5}241, 352\bar{4}1)$
$S_{POP} \circ Q$	$Av(2431, 4231, 4312)$
$S_{POP} \circ B$	$Av(2341, 2431, 3241, 3412, 4231, 4312)$
$Q_{POP} \circ S$	$Av(24513, 24531, 34251, 35241, 42513, 42531, 45231)$
$Q_{POP} \circ Q$	$Av(4321, 25413, 52413)$
$Q_{POP} \circ B$	$Av(3421, 4321, 24513, 25413, 52413, \img alt="Diagram of a 5x5 grid with a shaded top-left 2x2 corner and several black dots representing a permutation." data-bbox="718 631 788 686"/>$
$S \circ Q_{POP}$	$Av(2431, 4231, 23514, \img alt="Diagram of a 5x5 grid with a shaded top-left 2x2 corner and several black dots representing a permutation." data-bbox="666 688 746 738"/>$
$Q \circ Q_{POP}$	$Av(4321, 35214, 35241)$
$B \circ Q_{POP}$	$Av(2431, 4231, 4321, 23514, \img alt="Diagram of a 5x5 grid with a shaded top-left 2x2 corner and several black dots representing a permutation." data-bbox="688 761 768 811"/>$

Si ritiene interessante osservare anche la quantità di permutazioni di lunghezza n ordinabili dalle composizioni previste. La tabella che

segue mostra le combinazioni esaminate e la quantità di permutazioni di lunghezza n che esse ordinano per $n = 1 \dots 10$.

Composizione	1	2	3	4	5	6	7	8	9	10
$S \circ B$	1	2	6	20	70	252	924	3432	12870	48620
$Q \circ B$	1	2	6	22	90	394	1806	8558	41586	206098
$Q \circ S$	1	2	6	24	117	652	3988	26112	180126	1295090
$B \circ S$	1	2	6	22	89	380	1678	7584	34875	162560
$S \circ Q$	1	2	6	22	90	394	1806	8558	41586	206098
$B \circ Q$	1	2	6	21	79	311	1265	5275	22431	96900
$S_{\text{POP}} \circ S$	1	2	6	20	72	272	1064	4272	17504	72896
$S_{\text{POP}} \circ Q$	1	2	6	21	79	311	1265	5275	22431	96900
$S_{\text{POP}} \circ B$	1	2	6	18	54	162	486	1458	4374	13122
$Q_{\text{POP}} \circ S$	1	2	6	24	113	578	3094	17056	96093	550886
$Q_{\text{POP}} \circ Q$	1	2	6	23	101	480	2400	12434	66142	359112
$Q_{\text{POP}} \circ B$	1	2	6	22	86	342	1366	5462	21846	87382
$S \circ Q_{\text{POP}}$	1	2	6	22	88	368	1580	6904	30556	136582
$Q \circ Q_{\text{POP}}$	1	2	6	23	101	480	2400	12434	66142	359112
$B \circ Q_{\text{POP}}$	1	2	6	21	76	276	998	3589	12845	45803

Si vogliono individuare, a partire dalle righe di questa tabella, eventuali sequenze numeriche già note i cui primi 10 elementi coincidano con le quantità di elementi ordinabili da una combinazione di operatori. Si noti come il fatto che questi elementi coincidano con i primi 10 elementi di una sequenza nota non dà garanzia che le due sequenze siano equivalenti per $n > 10$. Si vuole semplicemente osservare e registrare queste somiglianze almeno per $n \leq 10$. Si è utilizzato la pagina web <https://oeis.org/> per individuare tali sequenze.

La sequenza data dalla quantità di permutazioni di lunghezza n ordinabili da $S \circ B$ al crescere di n ha gli stessi elementi per $n = 1 \dots 10$ della sequenza dei coefficienti binomiali centrali (A000984).

La sequenza A319027 ottenuta dalla combinazione $Q \circ S$ rappresenta esattamente il numero esatto di permutazioni la cui controimmagine tramite stacksort evita pattern 321, il che ci fornisce un'ulteriore prova della correttezza di tale risultato. Questo è l'unico caso in cui è lecito affermare che le due sequenze siano equivalenti.

La sequenza che ha i primi 10 elementi equivalenti al numero di permutazioni ordinabili da $B \circ S$ è A165543, che è composta dal numero esatto di permutazioni che evitano pattern 3241 e 4321.

La riga dedicata a $S_{POP} \circ S$ corrisponde ai primi 10 numeri della sequenza A071356, espansione di $\frac{1-2x-\sqrt{1-4x-4x^2}}{4x^2}$.

Da $S_{POP} \circ B$ si ottengono i primi 10 elementi della sequenza A025192, ottenuta dalla seguente ricorsione:

$$a(n) = \begin{cases} 1, & n=0 \\ 2 \cdot 3^{(n-1)}, & n \geq 1 \end{cases}$$

La combinazione $Q_{POP} \circ B$ ci suggerisce la sequenza A047849 $a(n) = (4^n + 2)/3$.

Si individuano alcune ricorrenze, ovvero alcune combinazioni ordinano la stessa quantità di permutazioni (anche se non le stesse). In particolare $Q \circ B$, $S \circ Q$, da cui si ottiene la sequenza A006318 *Large Schröder numbers*, e $B \circ Q$, $S_{POP} \circ Q$ che ordinano una quantità di permutazioni corrispondente ai primi 10 elementi della sequenza A033321 *Binomial transform of Fine's sequence*.

Un'ulteriore coppia di composizioni che ordinano lo stesso numero di permutazioni è data da $Q \circ Q_{POP}$, $Q_{POP} \circ Q$, composte dagli stessi due operatori. La sequenza che si ottiene da queste coincide con i primi 10 elementi della sequenza A218225, nota perché la sua funzione generatrice $A(x)$ soddisfa la seguente condizione:

$$\frac{1 - xA(x)}{1 - x^2A(x)^2} = 1 - x$$

Si osserva come le sequenze trovate da $Q_{POP} \circ S$, $S \circ Q_{POP}$, $B \circ Q_{POP}$ non corrispondono a nessuna sequenza nota.

APPENDICE: CODICE SORGENTE DEI PROGRAMMI REALIZZATI

PERMUTASORT

permutasort.py:

```
import sys
from itertools import permutations

if __name__ == '__main__':
    from src.operators import *
    from src.utils import *
else:
    from .src.operators import *
    from .src.utils import *

class selectorPermutations:
    def __init__(self,num,op):
        if num <=0:
            print("ERROR: was expecting a positive integer but got " +str(num))
            exit()

        # initialization of lists
        self.__sortable = []
        self.__unsortable = []
        self.__outcomes = []

        # generation of permutations
        permutations_list = list(permutations(range(1,num+1)))

        for P in permutations_list:

            # applying the operator to the permutation
            op_P_ = op(P)

            # adding outcome to the list
            if op_P_ not in self.__outcomes:
                self.__outcomes.append(op_P_)
```

```

    # adding permutations to the right list
    if isIdentityPermutation(op_P_):
        self.__sortable.append(P)
    else:
        self.__unsortable.append(P)

def getSortable(self):
    return self.__sortable

def getUnsortable(self):
    return self.__unsortable

def getOutcomes(self):
    return self.__outcomes

if __name__ == '__main__':

    if len(sys.argv) != 3:

        print("ERROR: expected 2 arguments but got these
              "+str(len(sys.argv)-1)+": " + str(sys.argv[1:]))
        exit()

    else:

        n = sys.argv[1]
        op = sys.argv[2]

        ps = selectorPermutations(int(n), getOperator(op))

        sortable = ps.getSortable()
        unsortable = ps.getUnsortable()
        outcomes = ps.getOutcomes()

        sortable_str = "The following " + str(len(sortable)) + "
                       "+str(n)+"-permutations are sortable with the operator "+op+":\n"
                       + printlist(sortable)
        unsortable_str = "The following " + str(len(unsortable)) + "
                       "+str(n)+"-permutations are not sortable with the operator
                       "+op+":\n" + printlist(unsortable)
        outcomes_str = "The operator "+op+" can give the following "+

```

```

    str(len(outcomes))+" results when applied to " + str(n) +
    "-permutations:\n" + printlist(outcomes)

```

```

print('\n'+sortable_str)
print(unsortable_str)
print(outcomes_str)

```

```

writeInFile("./log/"+n+op+"sortable.txt", sortable_str)
writeInFile("./log/"+n+op+"unsortable.txt", unsortable_str)
writeInFile("./log/"+n+op+"outcomes.txt", outcomes_str)

```

src/utls.py:

```

def printlist(list):
    result = ""
    for item in list:
        result = result +(str(item) + "\n")
    return result

def getOperator(key):
    from .operators import algorithmsIndex, compositionOf
    if len(key)==1 or (len(key)==2 and (key[0]=='P')):
        if key not in algorithmsIndex:
            print("ERROR: unsupported operator " + key)
            exit()
        return algorithmsIndex[key]
    else:
        if key[0]=='P':
            first_operator = key[:2]
            second_operator = key[2:]
        else:
            first_operator = key[:1]
            second_operator = key[1:]
        return
            compositionOf(getOperator(first_operator),getOperator(second_operator))

def writeInFile(name, content):
    file = open(name, 'w')
    file.write(str(content) + '\n')
    file.close()

def isIdentityPermutation(P):

```

```

n = 0
for p in P:
    n = n + 1
    if int(p)!=n:
        return False
return True

```

src/operators.py:

```

from .algorithms.bubblesort import B
from .algorithms.quuesort import *
from .algorithms.stacksort import *

#if implementing new operators, add them to the index
algorithmsIndex = {
    "B": B,
    "Q": Q,
    "S": S,
    "PS": PS,
    "PQ": Cons
}

def compositionOf(op1,op2):
    return lambda P: op1(op2(P))

```

NOTA: Il package algorithm contiene le implementazioni degli algoritmi bubblesort, stacksort, POP-stacksort, quuesort e POP-quuesort. Questi sono rappresentati da funzioni nominate rispettivamente B, S, PS, Q e Cons. Il dizionario algorithmsIndex è utilizzato per ottenere un operatore partendo dalla stringa che viene passata come argomento da linea di comando.

PATTFINDER

pattfinder.py:

```

import sys
from itertools import permutations, combinations

class PatternAvoid():
    def __init__(self, n, pattern):

```

```

if n <= 1:
    print("ERROR: expecting a whole number greater than 1 but got: "
          + str(n))
    exit()

self.__n = n

# checking if the argument really is a classical pattern
pattern_list = [int(p) for p in pattern]
if len(pattern) != max(pattern_list):
    print("ERROR: expected a pattern but got: " + str(pattern))
    exit()
for i in range(1, len(pattern)):
    if i not in pattern_list:
        print("ERROR: expected a pattern but got: " + str(pattern))
        print("\nA pattern of length n must have all the numbers from 1
              to n, " + str(i) + " not present")
        exit()

# generating permutations
permutations_list = list(permutations(range(1,n+1)))

# declaring list fields
self.__notcontaining = []
self.__containing = []

for p in permutations_list:
    if not self.contains(p,pattern):
        self.__notcontaining.append(p)
    else:
        self.__containing.append(p)

# get the standard image of the number sequence:
def patternize(self,pi):
    output = []
    id_ = sorted(pi)
    for p in pi:
        output.append(id_.index(p)+1)
    return output

# check if a sequence contains the given pattern
def contains(self, seq, pattern):

```

```

    if len(seq) < len(pattern):
        return False
    subseq = combinations(seq, len(pattern))
    for s in subseq:
        if list(self.patternize(s)) == list(pattern):
            return True
    return False

def getNotContaining(self):
    return self.__notcontaining

def getContaining(self):
    return self.__containing

def printlist(list):
    result = ""
    for item in list:
        result = result + (str(item) + "\n")
    return result

if __name__ == '__main__':

    if (len(sys.argv)==3):
        n = int(sys.argv[1])
        str_pattern = sys.argv[2]
        pattern = [int(char) for char in str_pattern]

        pa = PatternAvoid(n,pattern)

        con = pa.getContaining()
        ncon = pa.getNotContaining()

        ncon_str = "\nThe following "+str(len(ncon))+" " +
            str(n)+"-permutations do not contain the pattern " +
            str_pattern+":\n\n"+printlist(ncon)
        con_str = "The following "+str(len(con))+" " +
            str(n)+"-permutations do contain the pattern " +
            str_pattern+":\n\n"+printlist(con)

        print(ncon_str)
        print(con_str)

```

```
# only the not conatining get saved in a file
result_file = open("./log/"+str(n)+"Av"+str_pattern+".txt","w")
result_file.write(ncon_str+"\n")
result_file.close()

else:
    print("ERROR: unexpected number of arguments")
    exit()
```

BIBLIOGRAFIA

- [1] Michael H Albert, Mike D Atkinson, Mathilde Bouvel, Anders Claesson, and Mark Dukes. On the inverse image of pattern classes under bubble sort. *arXiv preprint arXiv:1008.5299*, 2010. (Cited on pages 14 and 24.)
- [2] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 11.)
- [3] Petter Brändén and Anders Claesson. Mesh patterns and the expansion of permutation statistics as sums of permutation patterns. *arXiv preprint arXiv:1102.4226*, 2011. (Cited on page 12.)
- [4] Lapo Cioni et al. Sorting with a popqueue. (Cited on pages 14, 32, and 33.)
- [5] Lapo Cioni and Luca Ferrari. Characterization and enumeration of preimages under the queuesort algorithm. In *Extended Abstracts EuroComb 2021: European Conference on Combinatorics, Graph Theory and Applications*, pages 234–240. Springer, 2021.
- [6] Lapo Cioni and Luca Ferrari. Preimages under the queuesort algorithm. *Discrete Mathematics*, 344(11):112561, 2021. (Cited on page 14.)
- [7] Anders Claesson and Henning Ulfarsson. Sorting and preimages of pattern classes. *Discrete Mathematics & Theoretical Computer Science*, (Proceedings), 2012. (Cited on pages 11, 23, 24, and 29.)
- [8] CONG HAN LIM. Brief introduction on stack sorting. (Cited on pages 9 and 14.)
- [9] Hjalti Magnússon. Sorting operators and their preimages. *Computer Science*, 2013. (Cited on pages 14 and 24.)
- [10] HENNING ÚLFARSSON. Ultra-quick tutorial for generalized pattern macros. <http://staff.ru.is/henningu/notes/patternmacros/patternmacros.tex>. (Cited on page 12.)