



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: *Relatore*  
Correlatore: *Correlatore*

Anno Accademico 2023-2024



*"Inserire citazione"*  
— *Inserire autore citazione*



---

INTRODUZIONE

---



---

## ALGORITMI DI ORDINAMENTO E CLASSI DI PATTERN

---

In questo capitolo verranno introdotti i principali algoritmi di ordinamento che utilizzano sorting devices, in particolare stack-sort, queue-sort e bubble sort, e altri concetti necessari per l'analisi della loro composizione.

Durante l'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.

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

Essendo interessati al comportamento di una sola iterazione di questi algoritmi si esaminerà un operatore, definito appositamente per ogni algoritmo, che descrive la singola iterazione.

Ad esempio, prendendo l'algoritmo bubble-sort si farà riferimento all'operatore  $B(\pi)$ , dove  $\pi$  è una permutazione di interi, tale che  $n$  iterazioni del bubble-sort possano essere rappresentate da  $B^n(\pi) = B(\dots B(\pi) \dots)$ .

### BUBBLE SORT

L'algoritmo di ordinamento bubble-sort prevede di scorrere gli elementi da ordinare dal primo al penultimo, ed ogni volta confrontare ogni elemento con il suo successivo per scambiarli se non sono ordinati.

Il risultato di una singola iterazione di bubble-sort su una permutazione  $\pi = \pi_1 \pi_2 \dots \pi_n$  è calcolato dall'operatore  $B(\pi)$ . Per una permutazione  $\pi$

con valore massimo  $n$  vale che  $\pi = \pi_L n \pi_R$ , allora  $B(\pi) = B(\pi_L) \pi_R n$ .

---

**Algorithm 1**  $B(\pi)$ 


---

```

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

```

---

### STACK SORT

L'operatore  $S(\pi)$  rappresenta il risultato ottenuto applicando un'iterazione di stack sort su una permutazione  $\pi$ .

Il primo passo consiste nell'inserire  $\pi_1$  nella pila. Poi lo si confronta con l'elemento  $\pi_2$ . Se  $\pi_1 > \pi_2$  allora il secondo viene messo nella pila sopra  $\pi_1$ , altrimenti  $\pi_1$  viene estratto dalla pila e inserito nell'output e  $\pi_2$  viene inserito nella pila.

Gli stessi passi vengono eseguiti per tutti gli altri elementi presenti nell'input, se viene trovato un elemento nell'input maggiore dell'elemento in cima alla pila, la pila viene svuotata finché questa condizione non diviene falsa, poi l'elemento viene spinto nella pila.

Finiti gli elementi nell'input, se necessario, si svuota completamente la pila nell'output.[7]

Sia  $\pi = \pi_L n \pi_R$ , con  $n$  valore massimo in  $\pi$ , vale che  $S(\pi) = S(\pi_L) S(\pi_R) n$

---

**Algorithm 2** operatore  $S$  - stack sort, singola iterazione

---

```

1: initialize an empty stack
2: for  $i = 1$  to  $n$  do
3:   while stack is unempty and  $\pi_i > \text{top of the stack}$  do
4:     pop from the stack to the output
5:   end while
6:   push( $\pi_i$ )
7: end for
8: empty the stack in the output

```

---

### QUEUE SORT

Per ogni elemento  $\pi_i$  della permutazione  $\pi$  in input se la coda é vuota o il suo ultimo elemento é minore di  $\pi_i$ , si accoda  $\pi_i$ , altrimenti si tolgono



elementi dalla coda ponendoli nell'output fino a che l'elemento davanti alla coda non è maggiore di  $\pi_i$ , poi si aggiunge  $\pi_i$  all'output. Si svuota la coda nell'output[8].

---

**Algorithm 3** operatore Q - queue sort, singola iterazione

---

```

1: initialize an empty queue
2: for  $i = 1$  to  $n$  do
3:   if empty queue or last in queue  $< \pi_i$  then
4:     enqueue( $\pi_i$ )
5:   else
6:     while first in queue  $< \pi_i$  do
7:       dequeue( $\pi_i$ )
8:     end while
9:     add  $\pi_i$  to the output
10:  end if
11: end for
12: empty the queue in the output

```

---

**BYPASS** L'operazione che pone un elemento nell'output senza passare dal contenitore si dice **bypass**. Questa viene svolta normalmente nel queuesort, ma talvolta può essere introdotta in altri algoritmi.

**OSSERVAZIONE** *Bubble sort* è un caso particolare sia di *queue sort* che di *stack sort*.

Se infatti si fissa a 1 la dimensione 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.

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

La nozione di *LTR maxima* risulta molto importante per lo studio di alcuni degli algoritmi di ordinamento presentati. In particolare durante l'esecuzione di *queuesort* i *LTR maxima* 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 stack sort e queue sort 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**

#### CLASSI DI PATTERN DI PERMUTAZIONI

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

Si dice che una permutazione  $\delta$  è un pattern contenuto in una permutazione  $\tau$  se esiste una sottosequenza di  $\tau$  di ordine isomorfico rispetto a  $\delta$ , e si indica con  $\delta \preceq \tau$ . Ad esempio 24153 contiene il pattern 312 perché  $413 \subset 24153$ .

**PATTERN STANDARD** Una **standardizzazione**[6] di una sequenza di numeri 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.

**CLASSI DI PATTERN** La relazione di sottopermutazione è una relazione di ordine parziale che viene studiata con dei sottoinsiemi chiamati **pattern di classi**. Ogni classe di pattern  $D$  può essere caratterizzata dall'insieme minimo  $M$  che evita:

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

**PATTERN BARRATI** I pattern classici esprimono in quale relazione di 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. La barra che viene posta sopra ad un elemento di un pattern indica che se una sequenza contiene un elemento in quella posizione allora essa non contiene il pattern.

Ad esempio  $1\bar{4}23 \not\preceq 162534$ , dato che  $1423 \preceq 2534$ , mentre  $1\bar{4}23 \preceq 146235$  dato che  $123 \preceq 146, 235$

#### ALGORITMI E PATTERN-AVOIDANCE

**PATTERN 231** Una permutazione  $\pi$  contiene un pattern 231 se  $231 \preceq \pi$ , ovvero se  $\exists a < b < c : \pi = \dots b \dots c \dots a \dots$ .

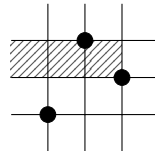
È noto in letteratura [7] che una permutazione può essere ordinata da una sola passata di stack sort se e solo se non contiene pattern 231.

Allo stesso modo, sono note simili condizioni perché una permutazione possa essere ordinata da una sola passata degli altri operatori descritti precedentemente.

Operatore	Permutazioni ordinabili con una sola passata
Stack sort	$Av(231)$
Queue sort	$Av(321)$
Bubble sort	$Av(231, 321)$
Pop-stack sort	$Av(231, 312)$
Pop-queue sort	$Av(321, 2413)$

#### MESH-PATTERN, PATTERN BARRATI E PATTERN DECORATI

In questa sezione verranno brevemente introdotti i mesh-pattern[3] attraverso degli esempi; le macro per la realizzazione delle griglie sono state prodotte dalla *Reykjavik University*[9].



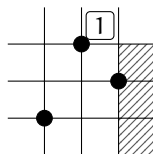
Il pattern mostrato nella griglia soprastante appare in una permutazione se si può individuare un pattern 132 posizionato in modo tale che le zone oscurate non sono occupate da altri elementi della permutazione.

La sequenza 1243 contiene un'istanza del pattern, dato che gli elementi 1, 4, 3 formano un pattern 132 e l'elemento 2 si trova in una zona non

oscurata.

La sequenza 3142 invece non contiene lo stesso pattern, dato che nonostante 1, 4, 2 formano un pattern 132, l'elemento 3 si trova nella zona oscurata (0, 2) (le zone oscurate si individuano indicando la posizione del loro angolo in basso a sinistra, numerandoli da 0).

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 é contenuto, ad esempio, nelle sequenze 1342, 12453 o 13524.

---

## PROGRAMMI REALIZZATI

---

### 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 é quello di enumerare tutte le  $n$ -permutazioni ordinabili e non ordinabili con una sola passata di  $X$ , stampa anche i possibili risultati di  $X$  su  $n$ -permutazioni.

La classe `SelectorPermutations` é 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 é ordinato la permutazione da cui si é ottenuto viene aggiunta alla lista `sortable`, altrimenti a `unsortable`. La classe fornisce dei *getters* per le liste e non ha altri metodi computativi, 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:

---

```
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

```

---

## 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 ottiene la versione ordinata della sequenza, e si sostituisce ogni valore con la posizione che occupa una volta ordinato;

contains verifica che una sequenza contenga 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.

---

```
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 lenght 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
```

---



---

## COMPOSIZIONE DI OPERATORI DI ORDINAMENTO

---

É molto interessante studiare la combinazione dei vari operatori e le relazioni tra classi di pattern e le loro preimmagini.

**PREIMMAGINI** Sia  $X$  un operatore di ordinamento, la preimmagine di un certo pattern  $p$ , secondo  $X$ , indicata con  $X^{-1}(p)$  rappresenta l'insieme di tutte le permutazioni la cui immagine secondo l'operatore  $X$  contiene il pattern  $p$ .

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

$Av(21)$  é 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 incrementali. Si indica con  $X^{-1}(Av(21))$  l'insieme di tutte le permutazioni ordinabili da  $X$ .

Ad esempio, dato che é noto che l'operatore *bubblesort* ordina solo le permutazioni che non contengono pattern 231 e 321, vale che:

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

**COMPOSIZIONE DI OPERATORI** Siano due operatori di ordinamento  $X$  e  $Y$ , la loro composizione é indicata con  $(XY) = (X \circ Y)$ , quindi, ad esempio, la composizione di *Stacksort* e *Bubblesort* (in questo ordine) si indica con  $SB(\pi) = (S \circ B)(\pi) = S(B(\pi))$ .

**ALGORITMI PER IL CALCOLO DI PREIMMAGINI** Sono già stati prodotti alcuni algoritmi per calcolare le preimmagini di pattern secondo gli operatori *bubblesort*[1], *stacksort*[6] e *queuesort*[8][5]. Questo ci permette, quando si combinano due operatori di ordinamento di cui del primo siano noti le classi di pattern da evitare per l'ordinabilità,

di cercare per quali pattern l'operatore che viene applicato per primo produce permutazioni che siano ordinabili dal secondo.

COMBINAZIONE DI *stacksort* E *bubblesort* Si considera dunque la composizione  $SB = S \circ B$  e ci si chiede quali permutazioni possano essere ordinate da esso.

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

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 preimmagini di *bubblesort*[1] si ottiene che:

$$(SB)^{-1}(Av(21)) = Av(3241, 2341, 4231, 2431)$$

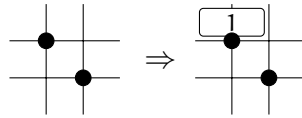
Per ottenere una preimmagine di *bubblesort* di un pattern classico, si applicano le seguenti regole:

si cercano tutte le coppie disordinate nell'immagine in esame: poiché *bubblesort* non produce nuove coppie disordinate, tutte le coppie disordinate nella preimmagine

si considera una lista di **candidati**, composta da ogni pattern minimo che contiene almeno le stesse coppie disordinate

per ogni candidato, si valuta ogni sua coppia disordinata  $(b, a)$ :

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



*Applicazione pratica per il calcolo di una preimmagine*

A titolo d'esempio verrà mostrato come si può arrivare al risultato (già noto) riguardo alla combinazione  $(S \circ B)$ :

$$(SB)^{-1}(Av(21)) = Av(2341, 2431, 3241, 4231)$$

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)
...
```

Questi risultati tuttavia sono utili come indicazione, ma non sono sufficienti per essere sicuri del risultato: infatti non escludono la presenza di pattern classici più lunghi né la presenza di pattern barrati.

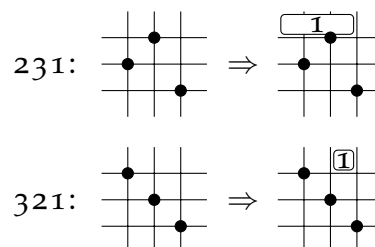
Per quanto quindi questo strumento si sia rivelato molto utile per avere un'idea di quali pattern ricercare, soprattutto durante le combinazioni più complesse che presenteremo dopo, non può sostituire del tutto un approccio più formale e teorico, come l'applicazione dell'algoritmo presentato prima, che viene mostrato di seguito:

Prima di tutto si cerca con quale operatore e di quale classe di pattern si deve ricercare la preimmagine:

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

Dato che, come noto  $S^{-1}(Av(21)) = Av(231)$ .

I pattern candidati ad essere preimmagini di  $231$  sono i pattern che contengono le coppie  $(2, 1)$ ,  $(3, 1)$ , ovvero i pattern  $231, 321$ . Si applicano le regole descritte prima:



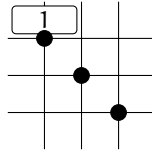
Quindi l'unica preimmagine secondo bubblesort di  $231$  che contiene  $321$  è  $3241$  mentre quelle che contengono  $231$  sono  $2341, 2431, 4231$ , che sono appunto i risultati che ci si aspettava.

$Q \circ B$ :

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

Si osserva che le coppie non invertite in 321 sono  $(3, 2), (3, 1), (2, 1)$  e l'unico pattern minimo che le contiene tutte é appunto 321.

Per fare in modo che *bubblesort* produca un 321 bisogna dunque che sia presente un valore 4 prima di 2: sia che sia posizionato prima di 3 o tra 3 e 2, esso é l'unico elemento (tra quelli che compongono il pattern) che viene spostato e il pattern 321 presente nell'input é ancora presente nell'output.



Cosí si ottengono le preimmagini 4321, 3421.

$$QB^{-1}(Av(21)) = Av(4321, 3421)$$

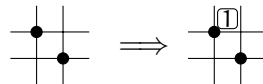
#### COMPOSIZIONI CHE TERMINANO CON *stacksort*

L'algoritmo per le preimmagini di *stacksort* é abbastanza simile a quello per *bubblesort*.

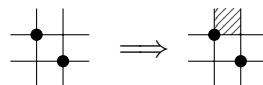
Anche in questo caso si osservano tutte le coppie disordinate contenute nell'immagine in esame e se ne tre una lista di pattern minimi candidati ad essere preimmagini e li si esaminano ad uno ad uno.

Per ogni coppia disordinata  $(b, a)$  presente nel candidato:

se  $(b, a)$  é 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 é presente allora si puó escludere la presenza di tale elemento



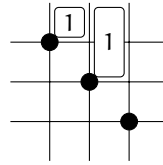
$Q \circ S$

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

Si ricerca dunque le preimmagini di 321 secondo *stacksort*.

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

L'elemento 3 deve entrare nella pila ed uscirne prima del 2, per fare ciò deve esserci un elemento  $3^+$  maggiore di 3 tra 3 e 2. Similmente é necessario un elemento  $2^+ > 2$  tra 2 e 1 per assicurare che 2 esca dalla pila prima che 1 vi entri.



Dunque le preimmagini che cerchiamo devono essere nella forma  $33^+22^+1$ .

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

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

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

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

$$(QS)^{-1}(Av(21)) = Av(34251, 35241, 45231)$$

$B \circ S$

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

Giá dall'analisi della combinazione precedente é risultato che  $S^{-1}(Av(321)) = Av(34251, 35241, 45231)$  quindi é necessario calcolare solo  $S^{-1}(Av(231))$ .

Quest'ultimo risultato é stato ampiamente studiato in letteratura, in quanto analogo al caso di una variante di *stacksort* che utilizza 2 pile. Il risultato che si ottiene é dunque che  $S^{-1}(Av(231)) = Av(2341, 3\bar{5}241)[6]$ .

Seguendo l'algoritmo applicato finora si osserva che i pattern candidati per le preimmagini sono 231, 321.

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 é minimo.

Inoltre i pattern  $35241, 3\bar{5}241$ , possono essere rappresentati dal pattern minimo  $3241$  che rende non minimo anche  $34251$ .

Il risultato che si ottiene é:

$$(BS)^{-1}(Av(21)) = Av(2341, 3241, 45231)$$

#### COMBINAZIONI CHE TERMINANO CON *queuesort*

Nonostante anche per *queuesort* sia stato trovato un algoritmo per le preimmagini[8], appprociare il problema "manualmente" analizzando i possibili comportamenti di *queuesort* rispetto alle diverse possibili permutazioni risulta essere piú semplice e piú comprensibile.

Se nel valutare il comportamento di *queuesort* rispetto ad una preimmagine se ci si vuole assicurare che un elemento entri nella coda allora lo si vuole "forzare" ad essere un LTR maxima, dunque si applicherá il seguente mesh-pattern all'elemento in considerazione:



Al contrario, se si ha a che fare con un elemento che risulta essere un right-to-left-maxima ma lo si vuole forzare a svuotare la coda, si avrà il seguente risultato:



Per forzare un elemento a effettuare un bypass é necessario che esso sia minore del primo elemento della coda, per questa condizione é piú difficile formalizzare una strategia, ma si vedranno in seguito esempi di come può essere fatto.

$S \circ Q$

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

Ancora una volta si considerano come candidati i pattern minimi che contengano almeno le coppie disordinate  $(2, 1), (3, 1)$  ovvero  $231, 321$ .

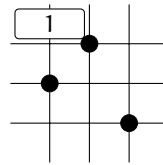
Se gli elementi del pattern  $231$  nell'immagine formano un pattern  $321$



nella preimmagine allora il valore 3 deve entrare nella coda. Sia che 2 effettui un bypass sia che provochi l'estrazione di alcuni elementi della coda comunque esso viene aggiunto all'output prima che 3 venga estratto. Si rende necessario un elemento  $3^+ > 3$  posto tra 2 e 1 per estarre il 3 dalla coda e inserirlo nell'input prima dell'1, tuttavia  $3^+$  non deve essere maggiore dell'elemento in fondo alla coda, altrimenti viene accodato. Si teorizza l'esistenza di un elemento  $4 > 3^+$  posto tra 3 e 2, che rappresenta il massimo della coda quando si raggiunge  $3^+$ .

Il pattern che si ottiene deve essere nella forma  $3423^+1$ , ovvero  $35241$

se invece gli elementi del pattern  $231$  nell'immagine formano un pattern  $231$  anche nella preimmagine che lo contiene a sua volta é necessario che 2 effettui un bypass o che venga estratto dalla coda prima che 3 vi entri. Il che implica la presenza di un valore  $2^+ > 2$  prima di 2 o tra 2 e 3. Similmente 3 deve essere aggiunto all'output prima di 1, quindi deve bypassare o essere estratto prima che 1 bypassi. Il che implica l'esistenza di un elemento 4 in posizione precedente a 1; in particolare i pattern minimi che rispettano queste condizioni sono quelli in cui 4 compare prima di 2 e si ha che  $2^+ = 4$ .



Gli unici pattern minimi che soddisfano queste condizioni sono  $2431, 4231$ .

Si nota inoltre che, il pattern calcolato prima  $35241 \in Q^{-1}(Av(321))$  risulta adesso superfluo in quanto  $4231 \preceq 35241$

$$(SQ)^{-1}(Av(21)) = Av(2431, 4231)$$

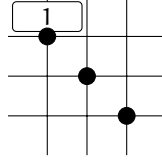
$B \circ Q$

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

Ancora una volta, si osserva che  $Q^{-1}(Av(231))$  é già stato calcolato nel caso precedente, e resta dunque da unire i risultati già ottenuti con  $Q^{-1}(Av(321))$ .

Gli elementi del pattern  $321$  nell'immagine devono essere a sua volta nello stesso ordine nell'immagine.

In particolare gli elementi 3, 2 devono effettuare un bypass o essere inseriti nella coda ed estratti prima dell'elemento successivo.



Il pattern 4321 sicuramente soddisfa queste condizioni: se 4 bypassa allora tutti gli elementi del pattern, essendo minori bypassano a loro volta, se viene inserito in coda ed estratto allora bypassano comunque, se viene inserito in coda e non estratto allora tutti gli elementi successivi evitano la coda e vengono aggiunti all'output, eventualmente dopo aver provocato l'estrazione di alcuni elementi dalla coda.

Se 3 viene aggiunto in coda ed estratto allora si ha una situazione 3421, dove 2 bypassa.

$$(SQ)^{-1}(Av(21)) = Av(2431, 3421, 4231, 4321)$$

#### CONTENITORI POP

Lo studio di combinazioni 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, é possibile individuare un caso di facile risoluzione: ovvero quello in cui un algoritmo che usa contenitori POP é concatenato ad un operatore regolare.

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

Sapendo l'insieme di pattern  $m$  che rendono la permutazione non ordinabile dall'operatore POP basterá cercare le loro preimmagini secondo l'operatore regolare con i metodi usati finora, come verrà mostrato nelle sezioni seguenti.

## POP QUEUESORT

L'algoritmo POP queuesort richiede una diversa analisi da POP stacksrt. 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 solo se il primo elemento della coda é il successivo dell'ultimo elemento aggiunto all'output; se l'elemento in input é maggiore dell'ultimo elemento della coda (o se la coda é vuota) allora viene accodato mentre negli altri casi, se l'elemento dell'input é 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; é la versione che verrà adottata in questa tesi; da qui in avanti ogni riferimento a *POP-queuesort* sarà riferito a *Cons*

---

**Algorithm 4** Cons - POP Queuesort
 

---

```

1:  $Q \leftarrow$  empty POP-queue
2: for  $i = 1$  to  $n - 1$  do
3:   if empty queue or  $\pi_i = \text{Back}(Q) + 1$  then
4:     enqueue( $\pi_i$ )
5:   else
6:     if  $\text{Front}(Q) > \pi_i$  then
7:       append  $\pi_i$  to the output
8:     else
9:       POP the queue and append the result to the output
10:      enqueue( $\pi_i$ )
11:    end if
12:  end if
13: end for
14: POP the queue and append the result to the output

```

---

La dimostrazione che *Cons* (cosí come *Min*) sia un algoritmo ottimale nella classe degli algoritmi *POP-queueosrt* si ha dal fatto che esso ordina tutte e sole le sequenze dell'insieme  $Av(321, 2413)$ , che sono esattamente tutte e sole le permutazioni ordinabili con una POP queue[4].

$S_{POP} \circ S$

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

Si é già calcolato che  $S^{-1}(Av(231)) = Av(2341, 3\bar{5}241)$ , quindi manca da calcolare le preimmagini di 312.

I pattern candidati che possono generare 312 sono quelli che contengono le coppie  $(3, 1)$ ,  $(3, 2)$ , ovvero 321, 312. Perché gli elementi di un pattern 321 nella preimmagine generino un pattern 312 tramite *stacksort* é necessario che 3 entri in pila e ne esca prima che 2 vi entri, deve quindi essere presente un valore 4 posizionato tra 3 e 2 che provochi l'uscita di 3, ovvero il pattern 3421.

Se gli elementi del pattern 312 nell'immagine formano lo stesso pattern anche nella preimmagine allora anche in questo caso bisogna assicurarsi che 3 venga estratto dalla pila prima di 1, si ha quindi il pattern 3412.

Si uniscono tutti i risultati:

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

$S_{POP} \circ B$

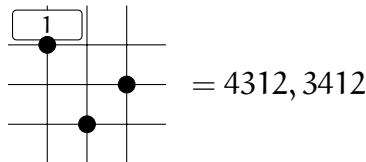
$$(S_{POP} \circ B)^{-1}(Av(21)) = B^{-1}(Av(231, 312))$$

Anche in questo caso la preimmagine di 231 é stata calcolata:  $B^{-1}(Av(231)) = Av(2341, 2431, 3241, 4231)$ .

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

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 é 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*:



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

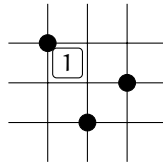
$$S_{\text{POP}} \circ Q$$

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

$$Q^{-1}(\text{Av}(231)) = \text{Av}(4231, 2431)$$

Possibili preimmagini di 312 : 312, 321.

Se la preimmagine contiene 312 bisogna far si che 3 effettui un bypass o che esca prima di 1, in ogni caso queste condizioni sono soddisfatte dal seguente mesh-pattern, 4312:



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

$$Q_{\text{POP}} \circ S$$

$$Q_{\text{POP}} \circ Q$$

$$Q_{\text{POP}} \circ B$$

IDEE PER LA RICERCA DI PREIMMAGINI SECONDO L'OPERATORE POP-STACKSORT

IDEE PER LA RICERCA DI PREIMMAGINI SECONDO L'OPERATORE POP-QUEUESORT

$$S \circ S_{\text{POP}}$$

$$Q \circ S_{\text{POP}}$$

$$B \circ S_{\text{POP}}$$

$$S \circ Q_{\text{POP}}$$

$$Q \circ Q_{\text{POP}}$$

$$B \circ Q_{\text{POP}}$$



---

## 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 17 and 18.)
- [2] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 8.)
- [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 9.)
- [4] Lapo Cioni et al. Sorting with a popqueue. (Cited on page 25.)
- [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. (Cited on page 17.)
- [6] Anders Claesson and Henning Ulfarsson. Sorting and preimages of pattern classes. *Discrete Mathematics & Theoretical Computer Science*, (Proceedings), 2012. (Cited on pages 8, 17, and 21.)
- [7] CONG HAN LIM. Brief introduction on stack sorting. (Cited on pages 6 and 9.)
- [8] Hjalti Magnússon. Sorting operators and their preimages. *Computer Science*, 2013. (Cited on pages 7, 17, and 22.)
- [9] HENNING ÚLFARSSON. Ultra-quick tutorial for generalized pattern macros. <http://staff.ru.is/henningu/notes/patternmacros/patternmacros.tex>. (Cited on page 9.)