



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.[5]

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  $n - 1$  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 é maggiore di  $\pi_i$ , poi si aggiunge  $\pi_i$  all'output. Si svuota la coda nell'output[6].

---

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

---

```

1: initialize an empty queue
2: for  $i = 1$  to  $n - 1$  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* é 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 é quello di una cella che, scorrendo l'input, contiene sempre il massimo valore trovato, mentre gli altri vengono messi nell'output.

**CONTENITORI POP** Un caso di studio interessante é quello in cui i contenitori di stack sort e queue sort vengano sostituiti dalla loro versione POP, cioè che 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  $\alpha$  é una sottosequenza di  $\beta$ , anche se non necessariamente una sottosequenza

consecutiva.

Si dice che una permutazione  $\delta$  é 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**[4] di una sequenza di numeri é un'altra sequenza della stessa lunghezza in cui l'elemento minore della sequenza originale é stato sostituito da 1, il secondo minore con un 2, ecc.

Ad esempio, la standardizzazione di 5371 é 3241.

**CLASSI DI PATTERN** La relazione di sottopermutazione é 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:

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

#### ALGORITMI E PATTERN-AVOIDANCE

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

É noto in letteratura [5] 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	$Av(132, 321)$

---

## 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

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

# more lines present in the original file are omitted
# from this one due to excessive lenght since it's not of our
# interest to show further in the thesis

```

---

## PATTFINDER

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

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

L'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

---

[Partire dalla combinazione di S e B [1][ok], poi spiegare l'algoritmo per S e B [4][ok] e mostarne l'applicazione per BS, QS, QB[ok], infine mostrare i passaggi per queuesort e mostrare BQ e SQ, facendo riferimento alla ricerca di preimmagini di pattern generici secondo Q [6]] É un caso molto interessante studiare la combinazione dei vari operatori e le relazioni tra determinati 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  $p$ .

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

Dato che  $Av(21)$  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*[4] e *queuesort*[6][3].

Questo ci permette, quando si combinano due operatori di ordinamento, 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 i suddetti algoritmi[1] si ottiene che:

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

Si può notare che bubble sort, applicato a tutte le combinazioni trovate, produce sempre un pattern 231:

2341: viene scambiato il 4 con l'1, ottenendo 2314

2431: viene scambiato il 4 con il 3, poi con l'1, ottenendo 2314

3241: vengono scambiati il 2 con il 3 e il 4 con l'1, ottenendo 2341

4231: viene scambiato il 4 con tutti gli elementi fino ad essere alla fine della sequenza.

Si nota da questo esempio un'approccio (senz'altro meno rigoroso degli algoritmi già formalizzati) che permette in generale di cercare preimmagini secondo bubblesort: dal pattern che si vuole ottenere si cercano le coppie di valori non ordinate (che quindi dovranno essere disordinate anche prima della passata) e si cercano le sequenze che le contengono; nel caso specifico del bubblesort, posizionando prima di una coppia disordinata nel pattern originale un valore maggiore, si evita che questa venga ordinata e questo ci permette di trovare le possibili preimmagini valutando le possibili posizioni e i possibili valori che questo può assumere.

**ESEMPIO PRATICO** Volendo ricercare le preimmagini di 231 secondo *bubblesort* si osserva che le coppie disordinate in 231 sono (2, 1), (3, 1), da questo si ricavano una lista di candidati, ovvero di pattern contenuti nelle preimmagini, che dovranno comprendere almeno le stesse coppie disordinate: ovvero 231 e 321.

231: *bubblesort* lascia invariata la coppia (2, 3), ma per produrre un pattern 231 é necessario che non venga scambiata la coppia (3, 1), per fare questo si posiziona il valore 4 prima dell'1, in qualunque posizione, ottenendo 4231, 2431, 2341;

321: per ottenere un 231 é necessario che venga scambiata la coppia (3, 2) ma non (3, 1), dunque é necessario un valore maggiore di 3 sia posizionato dopo 2 ma prima di 1, cosí si ottiene 3241.

Da queste considerazioni si puó verificare il risultato ottenuto prima per SB.

Con lo stesso metodo si puó trovare le preimmagini per le varie combinazioni che includono *bubblesort*:

*queuesort*  $\circ$  *bubblesort*:

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

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 4231, 3421.

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

#### COMPOSIZIONI 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 trae 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 può escludere la presenza di tale elemento (puó eventualmente essere necessario specificare nel pattern che questo elemento deve essere assente, il che può richiedere pattern non classici o meta-pattern[4])

---

## 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 15 and 16.)
- [2] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 7.)
- [3] 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 16.)
- [4] Anders Claesson and Henning Ulfarsson. Sorting and preimages of pattern classes. *Discrete Mathematics & Theoretical Computer Science*, (Proceedings), 2012. (Cited on pages 8, 15, 16, and 18.)
- [5] CONG HAN LIM. Brief introduction on stack sorting. (Cited on pages 6 and 8.)
- [6] Hjalti Magnússon. Sorting operators and their preimages. *Computer Science*, 2013. (Cited on pages 7, 15, and 16.)