



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.[4]

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 è maggiore di  $\pi_i$ , poi si aggiunge  $\pi_i$  all'output. Si svuota la coda nell'output[5].

---

**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* 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.

**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, 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  $\alpha$  è 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  $413 \subset 24153$ .

**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

É noto in letteratura [4] che una permutazione può essere ordinata da una sola passata di stack sort se e solo se non contiene pattern 231.

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

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	$Av(321, 2413)$
Pop-stack sort con bypass	$Av(231, 4213)$

---

## PROGRAMMI REALIZZATI

---

### PERMUTASORT

Il primo programma che ho realizzato viene lanciato 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()

self.__sortable = []
self.__unsortable = []
self.__outcomes = []

perms = list(permutations(range(1,num+1)))
for P in perms:
    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 isPermutationSorted(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

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

```

---

PATTFINDER



---

## COMPOSIZIONE DI OPERATORI

---

[Partire dalla combinazione di S e B [1], poi spiegare l'algoritmo per S e B [3] e mostarne l'applicazione per BS, QS, QB, infine mostrare i passaggi per queuesort e mostrare BQ e SQ, facendo riferimento alla ricerca di preimmagini di pattern generici secondo Q [5]]





---

## 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 page 13.)
- [2] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 7.)
- [3] Anders Claesson and Henning Ulfarsson. Sorting and preimages of pattern classes. *Discrete Mathematics & Theoretical Computer Science*, (Proceedings), 2012. (Cited on page 13.)
- [4] CONG HAN LIM. Brief introduction on stack sorting. (Cited on pages 6 and 8.)
- [5] Hjalti Magnússon. Sorting operators and their preimages. *Computer Science*, 2013. (Cited on pages 7 and 13.)