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

1

INTRODUZIONE

2

ALGORITMI DI ODRINAMENTO 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 definise 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 piú 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 esaminerá un operatore, definito appositamente per ogni algoritmo, che descrive la singola iterazione.
Ad esempio, prendendo l'algoritmo bubble-sort si fará riferimento all'operatore $B(\pi)$, dove π é 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 $\pi = \pi_1\pi_2\dots\pi_n$ é calcolato dall'operatore $B(\pi)$. Per una permutazione π

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

Algorithm 1 $B(\pi)$

```
for i = 1 to n − 1 do
    if π_i > π_{i+1} then
        Swap π_i and π_{i+1}
    end if
end for
```

STACK SORT

L'operatore $S(\pi)$ rappresenta il risultato ottenuto applicando un'iterazione di stack sort su una permutazione π.
Il primo passo consiste nell'inserire π_1 nella pila. Poi lo si confronta con l'elemento π_2. Se $\pi_1 > \pi_2$ allora il secondo viene messo nella pila sopra π_1, altrimenti π_1 viene estratto dalla pila e inserito nell'output e π_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 π, vale che $S(\pi) = S(\pi_L)S(\pi_R)n$

Algorithm 2 operatore S - stack sort, singola iterazione

```
initialize an empty stack
for i = 1 to n − 1 do
    while  stack is unempty and π_i >top of the stack do
        pop from the stack to the output
    end while
    push(π_i)
end for
emtpy the stack in the output
```

QUEUE SORT

Per ogni elemento π_i della permutazione π in input se la coda é vuota o il suo ultimo elemento é minore di π_i, si accoda π_i, altrimenti si tolgono elementi dalla coda ponendoli nell'output fino a che l'elemento davanti alla coda non é maggiore di π_i, poi si aggiunge π_i all'output. Si svuota la coda nell'output[8].

Algorithm 3 operatore Q - queue sort, singola iterazione

initialize an empty queue
for $i = 1$ **to** $n - 1$ **do**
    if empty queue **or** last in queue $< \pi_i$ **then**
        enqueue(π_i)
    else
        while first in queue $< \pi_i$ **do**
            dequeue(π_i)
        end while
        add π_i to the output
    end if
end for
emtpy 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 puó 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, ovvero un contenitore con politiche d estrazione e inserimento analoghe ma 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 PERMTUAZIONI

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

Si dice che una permutazione δ é un pattern contenuto in una permutazione τ se esiste una sottosequenza di τ di ordine isomorfico rispetto a δ, 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 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 puó essere caratterizzata dall'insieme minimo M che evita:

$$D = Av(M) = \{\beta : \mu \npreceq \beta \forall \mu \in M\}$$

PATTERN BARRATI I pattern classici esprimono in quale relazione di ordine devono essere gli elementi di una sequenza. In alcuni casi puó rivelarsi necessario dover esprimere informazioni ulteriori, come l'assenza di un elemento in una data posizone. Questa informazione puó essere fornita grazie all'utilizzo dei pattern barrati.
Un pattern barrato é un pattern in cui ogni numero puó essere barrato.
La barra che viene posta sopra ad un elemento di un pattern indica che se una sequenza contiene un elemento in quella posizone allora essa non contiene il pattern.
Ad esempio $1\bar{4}23 \npreceq 162534$, 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 π contiene un pattern 231 se $231 \preceq \pi$, ovvero se $\exists a < b < c : \pi = \ldots b \ldots c \ldots a \ldots.$

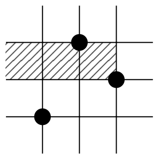
É noto in letteratura [7] che una permutazione puó essere ordinata da una sola passata di stack sort se e solo se non contiene pattern 231.

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)

MESH-PATTERN, PATTERN BARRATI E PATTERN DECORATI

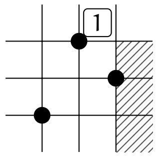
In questa sezione verranno brevemente introdotti i mesh-pattern[4], attraverso degli esempi, le macro per la realizzazione delle griglie sono state prodotte dalla *Reykjavik University*[1].



Il pattern mostrato nella griglia soprastante appare in una permutazione se si puó 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.

3

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 instanziata la classe viene usata come riferimento per i dati che ha giá 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
```

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 eventualemente 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 oridnato;

`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 seqence 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
```

4

COMPOSIZIONE DI OPERATORI DI ORDINAMENTO

É 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 identitá, 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 $(XY) = (X \circ Y)$, qundi, 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 giá stati prodotti alcuni algoritmi per calcolare le preimmagini di pattern secondo gli operaratori *bubblesort*[2], *stacksort*[6] e *queuesort*[8][5].
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 composione $SB = S \circ B$ 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[2] si ottiene che:

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

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

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

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

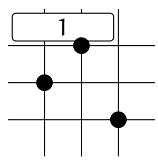
3241: vengono scambiati il 2 con il 3 e il 4 con l'1, ottenendo 2341

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

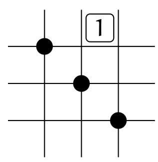
Si nota da questo esempio un'approccio (senz'altro meno rigoroso degli algoritmi giá formalizzati) che permette in generale di cercare preimmagini secondo bubblesort: dal pattern che si vuole ottenere si cercano le coppie di valori non ordinate (che quindi dovranno essere disordinate anche prima della passata) e si cercano le sequenze che le contengono, queste sequenze verranno dette **candidati** e ogni preimmagine del pattern deve contenere almeno uno di questi.
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 preimmaigini valutando le possibili posizioni e i possibili valori che questo puó assumere.

ESEMPIO DI CALCOLO DI UNA PREIMMAGINE 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 che un valore maggiore di 3 sia posizionato dopo 2 ma prima di 1, cosí si ottiene 3241.



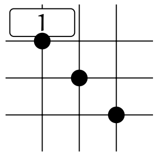
Da queste considerazioni si puó verificare il risultato ottenuto prima per SB.

Lo stesso metodo puó essere applicato alla combinazione di *queuesort* e *bubblesort*:

$Q \circ B$:

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

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.



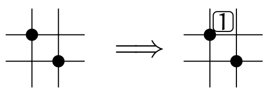
Cosí si ottengono le preimmagini 4321, 3421.

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

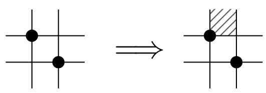
COMPOSIZIONI CHE TERMINANO CON *stacksort*

L'algoritmo per le preimmagini di *stacksort* é abbstanza 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



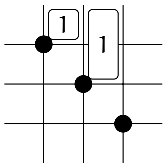
se invece (b, a) non é presente allora si puó escludere la presenza di tale elemento



Q ∘ 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 ció 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 $S^{-1}(Av(321)) = Av(34251, 35241, 45231)$ qundi 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 $S^{-1}(Av(231)) = Av(2341, 3\overline{5}241)$[6].

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

Per ottenere con *stacksort* un pattern 231 da un pattern 321 é necessario che il 3 entri pila e che vi rimanga fino all'ingresso di 2, il che esclude la presenza di valori maggiori di 3 tra 3 e 2, tuttavia dato che entrambi devono uscire dalla pila prima di 1 deve esserci almeno un valore maggiore di 3 tra 2 e 1: i pattern che implicano queste condizioni sono $3\overline{4}251$ e $3\overline{5}241$; si osserva che se i valori maggiori di 3 sono ordinati tra loro quando si arriva al secondo il primo viene estratto, contraddicendo le condizioni, dunque l'unico pattern minimo che contiene un pattern 321 e che genera un pattern 231 é $3\overline{5}241$

Se invece la preimmagine deve contenere e generare un pattern 231 si deve avere la presenza di un valore maggiore di 3 tra 3 e 1, che faccia uscire 3 dalla pila, ovvero il pattern 2341.

Si uniscono adesso i due insiemi ottenuti:

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

Si osserva che $2341 \preceq 34251$, quindi 34251 non é minimo.
Inoltre i pattern $35241, 3\overline{5}241$, 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], approcciare il problema "manualmente" analizzando i possibili comportamenti di *queuesort* rispetto alle diverse possibili permutazioni risulta essere piú semplice e piú comprensibile.

LEFT-TO-RIGHT-MAXIMA Nel valutare il comportamento di *queuesort* é importante considerare la nozione di *left-to-right maxima*, ovvero i valori in una sequenza che risultano maggiori di tutti i valori precedenti ad essi. Applicando *queuesort* ad una sequenza tutti i *left-to-rigth-maxima* vengono inseriti nella coda.

$S \circ Q$

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

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

Se la preimmagine ha un pattern 321 é necessario che l'elemento 3 entri nella coda per generare un pattern 231, questo si verifica se esso é un *left-to-right-maxima*.
2 viene sicuramente aggiunto all'output prima che 3 esca dalla coda, poi un valore maggiore di 3 deve essere presente prima di 1, per aggiungere 3 all'output prima che 1 effettui il bypass.
L'unico pattern che rispetta tutte queste condizioni é $\overline{5}3241$.

Per ottenere un'immagine che contenga un pattern 231 da una 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, qundi deve bypassare o essere estratto prima che 1 bypassi. Il che implica l'esistenza di un elemento 4 in posizione precdente a 1; si noti che se 4 compare prima di 2 il pattern minimo é dato da $2^+ = 4$.
Gli unici pattern minimi che soddisfano queste condizioni sono $2431, 4231$.
Anche i pattern $42351, 52341, 25341, 24351$ possono essere ottenuti dalle stesse condizioni, ma contengono tutti almeno uno dei due pattern elencati prima.

Si nota inoltre che, il pattern calcolato prima $\overline{5}3241 \in Q^{-1}(Av(321))$ risulta adesso superfluo in quanto $4231 \preceq \overline{5}3241$

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

$B \circ Q$

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

Ancora una volta, si osserva che $Q^{-1}(Av(231))$ é giá stato calcolato nel caso precedente, e resta dunque da unire i risultati giá ottenuti con $Q^{-1}(Av(321))$.
Si osserva che ogni candidato per la preimmagine del pattern 321 deve a sua volta contenere un pattern 321.
In particolare é necessario che tutti gli elemenri che compongono il pattern 321 effettuino un bypass. L'unico pattern minimo che soddisfa queste condizioni é 4321.
Si combina quest'ultimo risultato e con quelli del caso precedente e si ottiene:

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

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

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

$$P_{\mathrm{POP}}^{-1}(Av(21)) = Av(132, 321)$$

Si considerano due distinti casi: quello in cui l'operatore della combinazione applicato per primo alla permutazione é un algoritmo regolare e quello in cui é un algoritmo POP. Il primo caso é di piú facile soluzione: infatti si tratta di cercare le preimmagini dell'algoritmo regolare dei pattern che rendono la permutazione non ordinabile dall'algoritmo POP, seguendo le metodologie applicate finora (diversi risultati giá ottenuti per i casi regolari possono essere riutilizzati in questa circostanza).

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

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

Si é giá 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 un pattern 321 generi un 312 tramite *stacksort* é necessario che 3 entri in pila, ne esca prima che 2 vi entri, e che quando 2 vi entri non venga estratto prima di 1. Deve quindi essere presente un valore 4 posizionato tra 3 e 2 che provovhi l'uscita di 3, e bisogna assicurare che non vi sia un valore $2^+ > 2$ tra 2 e 1. Queste condizioni sono realizzate da i pattern nella forma $342\overline{2^+}1$, in cui bisogna valutare le condizioni di ordine di 2^+ rispetto agli elementi 3 e 4, differenziando tra i seguenti casi $(2^+ < 3), (3 < 2^+ < 4), (2^+ > 4)$ ed ottenendo i pattern $452\bar{3}1, 352\bar{4}2, 342\bar{5}1$.

BIBLIOGRAFIA

[1] (Cited on page 9.)

[2] 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.)

[3] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 8.)

[4] 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.)

[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.)