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 stacksort, queuesort e bubblesort, e altri concetti necessari per l'analisi ~~della loro composizione~~.

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

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

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 della i-esima iterazione i maggiori i elementi avranno raggiunto la loro posizione finale, dunque sono necessarie al massimo $n - 1$ iterazioni per ordinare la permutazione.

In questo capitolo si mostrerá il comportamento di una singola applicazione di questi algoritmi. Le proprietá trovate verranno poi sfruttate per lo studio di varie composizioni.
Infatti gereralmente quando si parla di questi algoritmi essi vengono descritti in funzione di varie iterazioni che vengono applicate fino a che la permutazione non é ordinata, mentre qui esamineremo gli effetti di una singola iterazione.

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

ALGORITMI

Bubblesort

L'algoritmo di ordinamento bubblesort prevede di scorrere dal primo al penultimo elemento e confrontarli ognuno con il proprio successivo per scambiarli se non sono ordinati.
Si definsce l'operazione $Swap(i, j)$, che fissata una permutazione π la resituisce con l'i-esimo e il j-esimo elementi scambiati di posizione.
Il risultato di una singola iterazione di bubblesort su una permutazione $\pi = \pi_1\pi_2 \dots \pi_n$ é calcolato dall'operatore B e si indica con $B(\pi)$.
Si puó scrivere permutazione π come $\pi_L n \pi_R$, dove n é il massimo valore della permutazione, π_L é il suo prefisso e π_R il suo suffisso. Vale che $B(\pi) = B(\pi_L)\pi_r n$.

Algorithm 1 $B - bubblesort$

```
1: for i = 1 to n − 1 do
2:     if π_i > π_{i+1} then
3:         Swap(i, i + 1)
4:     end if
5: end for
```

Stacksort

Stacksort é un algoritmo che utilizza una pila per ordinare una permutazione.
Si definisce una pila come un insieme di interi su cui si possono applicare le seguenti operazioni:

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

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

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

Per ogni elemento π_i presente nell'input se la pila é vuota viene eseguito un $\mathtt{push}$ e si passa al prossimo. Se la pila non é vuota si esegue l'operazione $\mathtt{pop}$ fino a che l'elemento in cima alla pila non é maggiore di π_i, poi si esegue un $\mathtt{push}$.
Finiti gli elementi nell'input, se necessario, si svuota completamente la pila nell'output.[8]
L'espressione $S(\pi)$ rappresenta il risultato ottenuto applicando un'iterazione di stacksort su una permutazione $\pi = \pi_1\pi_2\ldots\pi_n$.
In questo caso, sia $\pi = \pi_L n \pi_R$, con n valore massimo in π, e π_L, π_R rispettivamente suo prefisso e suffisso, vale che $S(\pi) = S(\pi_L)S(\pi_R)n$

Algorithm 2 S - stacksort

```
1: S ← ∅
2: for i = 1 to n do
3:     while S ≠ ∅ and π_i > Top(S) do
4:         Pop(S)
5:     end while
6:     Push(π_i, S)
7: end for
8: while S ≠ ∅ do
9:     Pop(S)
10: end while
```

Queue sort

Queuesort utilizza una coda per ordinare una permutazione.
Si definisce una coda come un insieme su cui si possono applicare le seguenti operazioni:

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

$\mathtt{Dequeue}(Q)$ estrae da Q l'elemento che é stato inserito per primo e lo aggiunge all'output

$\mathtt{Front}(Q)$ permette di osservare del prossimo valore da estarre senza rimuoverlo dalla pila

$\mathtt{Back}(Q)$ permette di osservare l'ultimo valore inserito nella pila.

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

$\mathtt{Bypass}(\pi_i)$ il valore π_i viene aggiunto all'output senza passare dalla coda

Per ogni elemento π_i della permutazione π in input se la coda 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[9].

Algorithm 3 operatore Q - queue sort, singola iterazione

```
1: Q ← ∅
2: for i = 1 to n do
3:     if Q = ∅ or Back(Q) < π_i then
4:         Enqueue(π_i,Q)
5:     else
6:         while Front(Q) < π_i do
7:             Dequeue(π_i)
8:         end while
9:         Bypass(π_i)
10:    end if
11: end for
12: while Q ≠ ∅ do
13:    Dequeue(π_i)
14: end while
```

OSSERVAZIONE *bubblesort* é un caso particolare sia di *queue sort* che di *stacksort*.
Se infatti si fissa a 1 la capacitá 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, ovvero analogo a quello di bubblesort.

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

La nozione di *LTR massimi* risulta molto importante per lo studio di alcuni degli algoritmi di ordinamento presentati. In particolare durante l'esecuzione di *queuesort* i LTR massimi sono tutti e soli gli elementi che

entrano nella coda, mentre durante l'esecuzione di *bubblesort* sono gli unici che vengono scambiati; in entrambi gli algoritmi l'ordine relativo degli altri elementi non viene alterato.

CONTENITORI POP Un caso di studio interessante é quello in cui i contenitori di stacksort 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

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

STANDARDIZZAZIONE Un pattern classico é sempre rappresentato dalla **standardizzazione** di una permutazione.
Per una sequenza di numeri la sua standardizzazione[7] 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 .

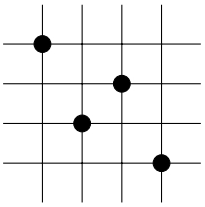
Si dice che una permutazione τ contine un pattern δ se esiste $\lambda \subseteq \tau$ la cui standardizzazione é uguale a δ, e si indica con $\delta \preceq \tau$.
Ad esempio 24153 contiene il pattern 312 perché $413 \subseteq 24153$ e la standardizzazione di 413 é 312.

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 D puó essere caratterizzata dall'insieme ~~minimo~~ M che evita:

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

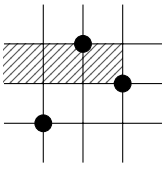
In questa sezione verranno da qui in poi introdotti brevemente altri tipi di pattern[3] attraverso degli esempi; le macro per la realizzazione delle griglie sono state prodotte dalla *Reykjavik University*[10].

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



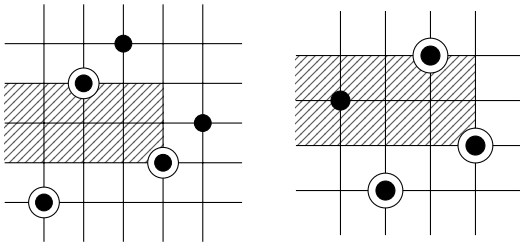
PATTERN BARRATI I pattern classici esprimono in quale relazione di ordine devono essere gli elementi di una sequenza. In alcuni casi puó rivelarsi necessario dover esprimere informazioni ulteriori, come l'assenza di un elemento in una data posizone. Questa informazione puó essere fornita grazie all'utilizzo dei pattern barrati.
Un pattern barrato é un pattern in cui ogni numero puó essere barrato. Un istanza di un pattern barrato é rappresentata da un'istanza della parte non barrata del pattern che non si estenda ad un occorrenza della parte barrata. Ad esempio una permutazione π contiene il pattern $23\overline{4}1$ se contiene degli elementi $a < b < c$ tali che $\pi = \dots b \dots c \dots a \dots$ e fra c e a non é presente un elemento maggiore di c.
Una permutazione quindi evita un pattern barrato se ad ogni occorrenza della parte non barrata del pattern corrisponde un'occorrenza della parte non barrata.

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

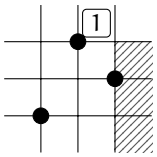


Una permutazione π di lunghezza n contiene questo pattern se ci sono 3 valori $a < b < c < n$ tali che $\pi = \dots a \dots c \dots b \dots$ e prima di b nessun valore é compreso tra b e c.

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



PATTERN DECORATI 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.

ALGORITMI DI ORDINAMENTO E PATTERN-AVOIDANCE

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

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

3

PROGRAMMI REALIZZATI

In questo capitolo presenteró il funzionamento di alcuni programmi che ho realizzato come supporto allo studio degli argomenti di questa tesi.

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

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 eventualemente 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 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
```

SCRIPT PER LA VERIFICA DEI PATTERN

Spesso ho utilizzato questo ulteriore script, che utilizza le classi presentate prima, per verificare che i pattern trovati per un dato operatore siano sufficienti a definire tutto l'insieme di permutazioni non ordinabili.

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

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

Di seguito un esempio di utilizzo ed il codice:

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

Generating permutations:
      found 50 unsortable permutations

Verifying 3 patterns:

Verifying pattern 2341:
      found 17 matches

Verifying pattern 2431:
      found 17 matches

Verifying pattern 3241:
```

```
      found 17 matches

These 10 permutations are not sorted by SB but not contain any
    specified pattern:
(5, 2, 4, 1, 3)
(5, 3, 1, 4, 2)
(5, 2, 3, 1, 4)
(5, 3, 4, 2, 1)
(1, 5, 3, 4, 2)
(5, 1, 3, 4, 2)
(4, 5, 2, 3, 1)
(4, 2, 3, 1, 5)
(5, 4, 2, 3, 1)
(5, 3, 4, 1, 2)

$ python verify_patterns.py SB 5 2341 2431 3241 4231

Generating permutations:
      found 50 unsortable permutations

Verifying 4 patterns:

Verifying pattern 2341:
      found 17 matches

Verifying pattern 2431:
      found 17 matches

Verifying pattern 3241:
      found 17 matches

Verifying pattern 4231:
      found 17 matches

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

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

# allows to find out if the specified patterns cover all the
    permutations of lenght n
```

```
# that are not sortable by the specified sorting operator

if __name__ == '__main__':

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

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

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

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

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

4

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 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*[1], *stacksort*[7] e *queuesort*[9][5].
Questo ci permette, quando si combinano due operatori di ordinamento (di cui di quello applicato per secondo siano noti le classi di pattern da evitare per l'ordinabilitá) di cercare per quali pattern l'operatore che

viene applicato per primo produce permutazioni che siano ordinabili dal secondo.

UN CASO GIÁ NOTO: 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 l'algoritmo per le preimmagini di bubblesort[1] si ottiene che:

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

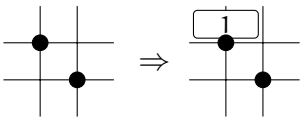
Per ottenere una preimmagine di bubblesort di un pattern classico, si applicano le seguenti regole:

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



Esempio pratico di calcolo di una preimmagine

A titolo d'esempio verrá mostrato come si puó arrivare al risultato (giá 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, puó aiutare ad arrivare alle stesse conclusioni. Infatti lanciando il programma sull'operatore SB e con $n = 4$ si ottengono i seguenti risultati:

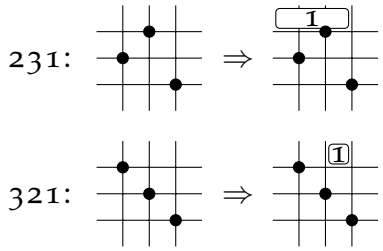
```
$ python permutasort.py 4 SB
...
The following 4 4-permutations are not sortable with the operator
SB:
(2, 3, 4, 1)
(2, 4, 3, 1)
(3, 2, 4, 1)
(4, 2, 3, 1)
...
```

Questi risultati tuttavia sono utili come indicazione, ma non sono sufficienti per essere sicuri del risultato: infatti non escludono la presenza di pattern classici piú 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 piú complesse che presenteremo dopo, non puó sostituire del tutto un approccio piú formale e teorico, come l'applicazione di uno degli algoritmi presentati prima, che viene mostrata 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:

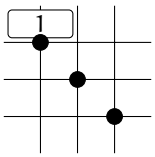


Quindi l'unica preimmagine secondo bubblesort di 231 che contiene 321 é 3241 mentre quelle che contengono 231 sono 2341, 2431, 4231, che sono appunto i risultati che ci si aspettava.

$Q \circ B$:

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

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.



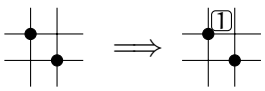
Cosí si ottengono le preimmagini 4321, 3421.

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

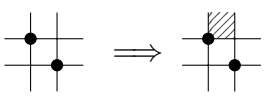
COMPOSIZIONI CHE TERMINANO CON *stacksort*

L'algoritmo per le preimmagini di *stacksort* é 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



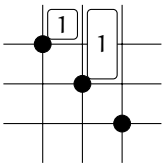
se invece (b, a) non é presente allora si puó escludere la presenza di tale elemento



$Q \circ S$

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

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 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 $S^{-1}(Av(231)) = Av(2341, 3\bar{5}241)$[7].

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[9], approcciare 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-lrft-maxima ma lo si vuole forzare a svuotare la coda, si avrá 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 startegia, ma si vedranno in seguito esempi di come puó essere fatto.

$S \circ Q$

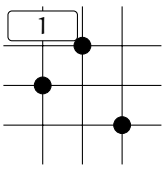
$$(SQ)^{-1}(Av(21)) = Q^{-1}S^{-1}(Av(21)) = Q^{-1}(Av(231))$$

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, qundi deve bypassare o essere estratto prima che 1 bypassi. Il che implica l'esistenza di un elemento 4 in posizione precdente a 1; in particolare i pattern minimi che rispettano queste condizoni 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 $4231 \preceq 35241$

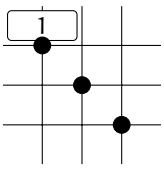
$$(SQ)^{-1}(Av(21)) = Av(2431, 4231)$$

$B \circ Q$

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

Ancora una volta, si osserva che $Q^{-1}(Av(231))$ é giá stato calcolato nel caso precedente, e resta dunque da unire i risultati giá ottenuti con $Q^{-1}(Av(321))$.
Gli elementi del pattern 321 nell'immagine devono essere a sua volta nello stesso odrine 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 verrá mostrato nelle sezioni seguenti.

POP QUEUESORT

L'algoritmo POP queuesort richiede una diversa analisi da POP stacksort. 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 verrá adottata in questa tesi; da qui in avanti ogni riferimento a *POP-queuesort* sará riferito a *Cons*

Algorithm 4 Cons - POP Queuesort

```
1:  Q ← empty POP-queue
2:  for i = 1 to n − 1 do
3:      if empty queue or π_i = Back(Q) + 1 then
4:          enqueue(π_i)
5:      else
6:          if Front(Q) > π_i then
7:              append π_i to the output
8:          else
9:              POP the queue and append the result to the output
10:             enqueue(π_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].

$S_{POP} \circ S$

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

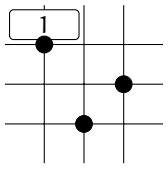
Si é giá calcolato che $S^{-1}(Av(231)) = Av(2341, 3\bar{5}241)$, quindi manca da calcolare le preimmagini di 312.
I pattern candidati che possono generare 312 sono quelli che contengono le coppie $(3,1), (3,2)$, ovvero $321, 312$. Perché gli elementi di un pattern 321 nella preimmagine generino un pattern 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:

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

$S_{POP} \circ B$

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

Anche in questo caso la preimmagine di 231 é stata calcolata: $B^{-1}(Av(231)) = (S \circ B)^{-1}(Av(21)) = Av(2341, 2431, 3241, 4231)$.
Si ricerca dunque le preimmaigini di 312 secondo bubblesort.
Gli elementi del pattern 312 nell'immagine possono solo formare un pattern 312 o 321 nella preimmagine.
Si puó 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*:



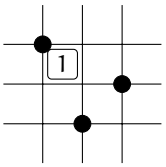
$= 4312, 3412$

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

$S_{POP} \circ Q$

$$(S_{POP} \circ Q)^{-1}(Av(21)) = Q^{-1}(Av(231, 312))$$
$$Q^{-1}(Av(231) = Av(4231, 2431)$$

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:



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

$Q_{POP} \circ S$

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

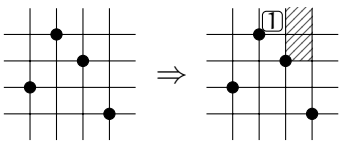
Come in molti altri casi la preimmagine di 321 é stata calcolata:

$$S^{-1}(Av(321)) = (QS)^{-1}(Av(21)) = Av(34251, 35241, 45231)$$

Si ricerca dunque le preimmagini di 2413 secondo stacksort, applicando lo stesso algoritmo dei casi precedenti.
Le coppie non ordinate di 2413 sono $(2, 1), (4, 1), (4, 3)$, che definiscono l'insieme di candidati 2413, 2431, 4213, 4231.

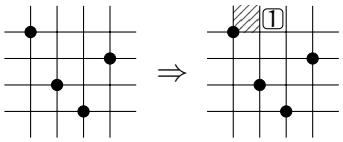
2431 I primi due elementi del pattern sono ordinati fra loro, dunque per lasciare il pattern invariato é sufficiente evitare che le coppie 41 e 43 vengano ordinate. Come giá visto questo si ottiene posizionando in mezzo ai due elementi della coppia un elemnto maggiore di entrambi. Si ottiene cosí il pattern 24513.

2431 Le coppie non ordinate di questa preimmagine sono quelle giá individute nell'immagine piú $(3, 1)$. Quindi applicando le condizioni per le preimmagini si ottiene il seguente meta pattern:



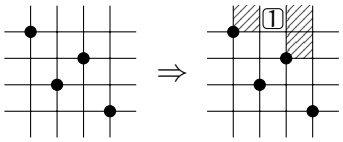
La preimmagine di 2413 che contiene 2431 é quindi 24531.

4213



La preimmagine di 2413 che contiene 4213 é 42513.

4231



La preimmagine di 4213 é 42531.

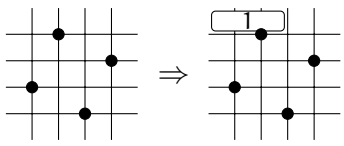
Unendo tutti i risultati si ottiene:

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

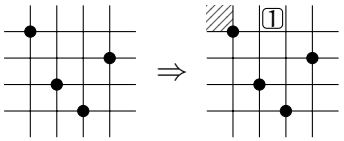
$Q_{POP} \circ B$

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

Come in tutti gli altri casi POP, una delle condizioni é giá stata calcolata: $B^{-1}(Av(321)) = (Q \circ B)^{-1}(Av(21)) = Av(3421, 4321)$. Si ricerca adesso le preimmagini di 2413 (secondo bubblesort).
I candidati per le preimmagini sono: 2413, 2431, 4213, 4231, 4321.
Si osserva subito che 2431, 4231, e 4321, pur essendo nell'elenco dei candidati non possono in realtá generare un pattern 2413 con gli stessi elementi: in ognuno di questi infatti la coppia $(3, 1)$ dovrebbe essere ordinata ma nessuno dei due é un LTR maxima, dunque il loro ordine relativo non viene cambiato da bubblesort.
Si applicano le condizioni per la ricerca di preimmagini di bubblesort agli altri due candidati:

2413: 

Dunque i pattern che contengono un pattern 2413 che resta invariato dopo una passata di bubblesort sono 24513, 25413, 52413.

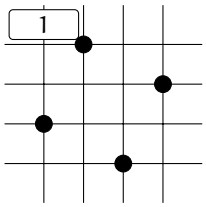
4213 

Il pattern minimo che corrisponde al mesh pattern rappresentato é 42513.
Si noti che l'area oscurata dovrebbe generare i pattern $\overline{6}42513$ e $\overline{5}42613$, tuttavia si nota che entrambi i pattern, se inclusa la parte barrat, sono contenuti in quello precedente e generano pattern 2413 dopo una passata di bubblesort: $B(642513) = B(542613) = 425136$, dove la sottosequenza 2513 rappresenta un'istanza del pattern 2413.

$Q_{POP} \circ Q$

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

La preimmagine $Q^{-1}(Av(321))$ é giá stata trovato nel caso $B \circ Q$, e vale che $Q^{-1}(Av(321)) = Av(4321)$.
Se si applica le regole per la ricerca di preimamgini di queuesort a 2413 si ottiene:



Quindi si ottiene che $(Q_{POP} \circ Q)^{-1}(Av(21)) = Av(4321, 25413, 52413)$.

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

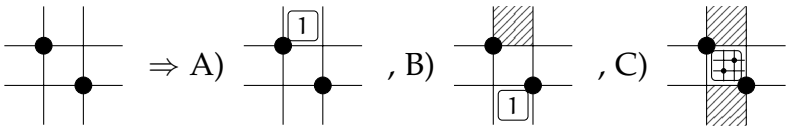
Si ricercano le condizioni che un candidato deve soddisfare per produrre una voluta immagine. Come negli altri casi, si esaminano le coppie non ordinate nel candidato, per ogni coppia disordinata (b, a) che non viene ordinata da una passata di POPstacksort si verifica una delle sequenti condizioni:

A) un elemento $c > b$ é presente tra b e a, questo provoca il POP prima che l'algoritmo arrivi ad esaminare a;

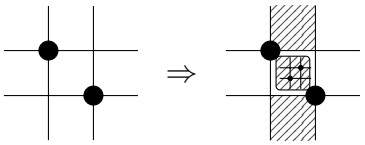
B) un elemento $a^- < a$ (e nessun elemento $c > b$) é presente tra b e a, quindi a^- viene inserito in coda ed é a a provocare il POP;

c) tra i due elementi ma non c'é nessun elemento minore di a o maggiore b ma c'é una coppia ordinata, ovvero esistono $b^-, a^+ : a < a^+ < b^- < b$ e $b \dots a^+ \dots b^- \dots a$ é una sottosequenza nel candidato.

Quindi per trovare una preimmagine secondo POPstacksort occorre applicare le seguenti decorazioni, per ogni coppia disordinata che debba restare tale, al meta pattern del candidato:



Se invece ci si vuole assicurare che una coppia disordinata venga ordinata, la decorazione da apllicare consiste semplicemente nella "negazione" di tutte quelle espresse finora:

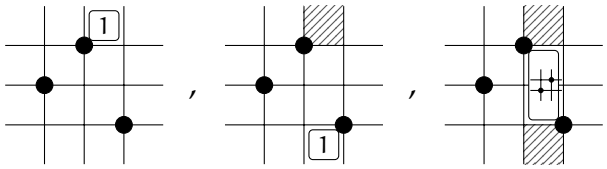


$S \circ S_{POP}$

$$(SS_{POP})^{-1}(Av(21)) = S_{POP}^{-1}(S^{-1}(Av(21))) = S_{POP}^{-1}(Av(231))$$

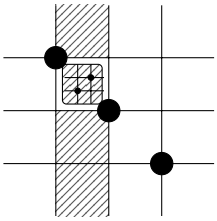
Si applicano le condizioni per le preimmagini ai due candidati: 231 e 321.

Nel caso di 231, le coppie disordinate sono $(2,1), (3,1)$. Si osserva che 3 giá soddisdfa la condzione A per la coppia $(2,1)$, quindi resta da applicare le condizioni alla coppia $(3,1)$:

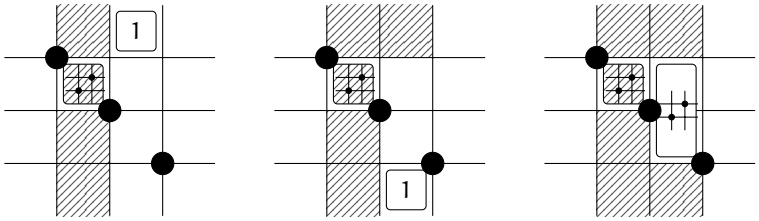


Dalla condizione A si ottiene il pattern 2341, dalla B il 3412 dalla condizione C si ottengono, considerando la posizione relativa degli elemeneti nella coppia aggiuntiva rispetto all'elemento 2, i pattern 25341, 35241, 45231.

Se gli elementi del pattern 231 nell'immagine invece formano un pattern 321 nella preimmagine, allora si applica la condizione per ordinare una coppia alla coppia $(3,2)$:



Si applicano adesso le condizioni per la coppia $(3,1)$ che ci si aspetta resti disordinata:



Si osservi come applicando le condzioni A e B per la coppia $(3,1)$ queste siano giá soddisfatte per la coppia $(2,1)$. Applicando la condizione C la coppia puó essere costruita utilizzando l'elemento 2 come membro minore all'interno della coppia, e questo soddisfa anche la condizione A per la coppia $(2,1)$. Si ottengono dunque i pattern 3241, 4312, 4231.

Si osserva che, tra i risultati ottenuti prima, alcuni risultano non necessari: $2341 \preceq 25341, 3241 \preceq 35241, 4231 \preceq 45231$. Si semplificano e si ottiene il seguente risultato:

$$(SS_{POP})^{-1}(Av(21)) = Av(2341, 3241, 3412, 4231, 4312)$$

$Q \circ S_{POP}$

$$QS_{POP}{}^{-1}(Av(21)) = S_{POP}^{-1}(Q^{-1}(Av(21))) = S_{POP}^{-1}(Av(321))$$

Si applicano anche stavolta le condizioni per la ricerca di preimmagini al pattern 321. Essendo 321 una sequenza decrementale, non ci sono altri candidati; le sue coppie disordinate sono $(3,2), (3,1), (2,1)$ ed é necessario applicare tutte le combinazioni possibili di condizioni A,B,C a queste coppie.

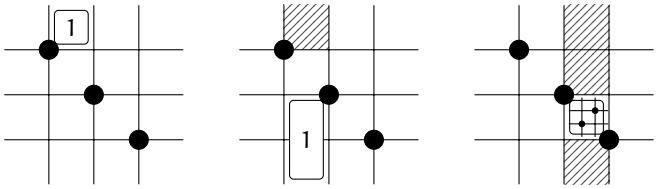
Una volta applicate le diverse condizioni alla coppia $(3,2)$ é evidente che queste soddisfano anche le condizioni per la coppia $(3,1)$:

applicando la condizione A e ponendo quindi un elemento maggiore di 3 tra 3 e 2, allora lo stesso elemento rappresenta un valore maggiore di 3 posizionato tra 3 e 1 (condzione A per la coppia $(3,1)$)

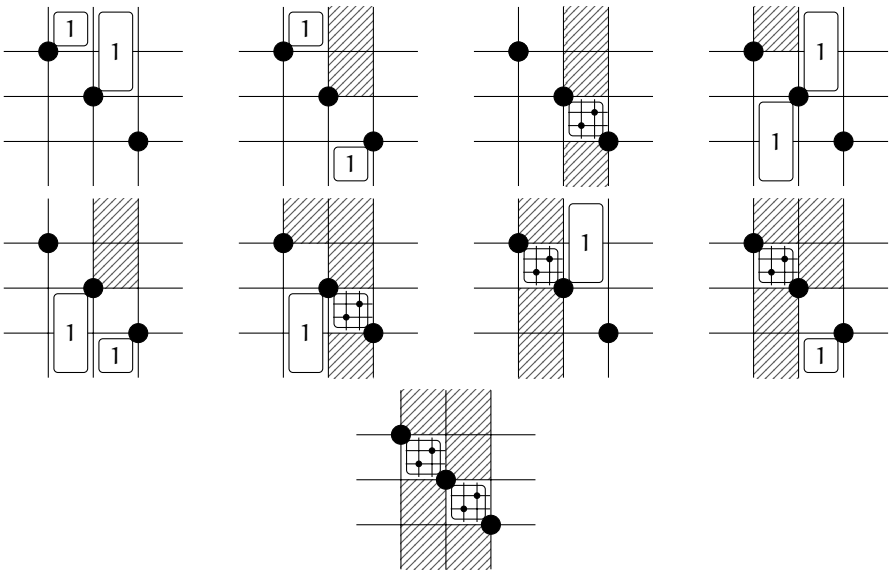
se si applica la condizione B, ponendo un valore $2^- < 2$ tra 2 e 3, allora o si ha che $2^- < 1$, che soddisfa la condizione B per la coppia $(3,1)$, o che $1 < 2^- < 2$, e allora la coppia $(2^-,2)$ é una coppia ordinata tra 3 e 1 (condizione C per ls coppia $(3,1)$)

nel caso, infine, in cui si applichi la condizione C e si ponga quindi una coppia ordinata tra 3 e 2 allora la stessa coppia soddisfa la condizione C per la coppia $(3,1)$.

In sintesi, i seguenti pattern soddisfano le condzioni da imporre alle coppie $(3,2),(3,1)$:



Una volta applicate le condizioni anche alla coppia $(2,1)$ si ottengono i seguenti 9 meta-pattern, da cui si ottengono 22 pattern classici:



Scrivendo come pattern classici quelli che si ottengono da questi meta-patterns ed eliminando quelli ridondanti, si ottiene il seguente risultato:

$$\mathrm{QS}_{\mathrm{POP}}(Av(21)) = Av(34251, 35241, 41352, 42351, 45231,$$
$$45312, 51342, 51423, 52341, 52413, 53412)$$

$B \circ S_{POP}$

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

Il l'insieme minimo di pattern per l'ordinabilitá di permutazione con la composzione bubblesort, POPstacksort é l'unione degli insiemi dei due risultati precedenti, stacksort e queuesort.
Si nota, tuttavia che ogni pattern contenuto nell'insieme ottenuto dalla combinazione con queuesort contiene **almeno** un pattern tra quelli ottenuti da stacksort. Nello specifico, a titolo esemplificativo:

$$3241 \preceq 34251, 35241, 41352, 42351$$

$$3412 \preceq 53412$$

$$4231 \preceq 45231, 51342, 52341, 52413$$

$$4312 \preceq 45312, 51423$$

Dunque l'insieme minimo di pattern che caratterizza le permutazioni ordinabili da $B \circ S_{POP}$ é lo stesso di $S \circ S_{POP}$.

$$(BS_{POP})^{-1}(Av(21)) = Av(2341, 3241, 3412, 4231, 4312)$$

$S \circ Q_{POP}$

[Congettura per la preimmagine di 231 (vedere file cons.pdf, poi copiare da cons.tex)]

$Q \circ Q_{POP}$

[Congettura per la preimmagine di 321, per ora scritta solo su carta]

$B \circ Q_{POP}$

[Unire risultati dei due paragrafi precedenti]

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 11, 21, and 22.)

[2] Mathilde Bouvel, Lapo Cioni, and Luca Ferrari. Preimages under the bubblesort operator. *arXiv preprint arXiv:2204.12936*, 2022. (Cited on page 9.)

[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 pages 11 and 29.)

[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 21.)

[6] Lapo Cioni and Luca Ferrari. Preimages under the queuesort algorithm. *Discrete Mathematics*, 344(11):112561, 2021. (Cited on page 11.)

[7] Anders Claesson and Henning Ulfarsson. Sorting and preimages of pattern classes. *Discrete Mathematics & Theoretical Computer Science*, (Proceedings), 2012. (Cited on pages 9, 21, and 25.)

[8] CONG HAN LIM. Brief introduction on stack sorting. (Cited on pages 7 and 11.)

[9] Hjalti Magnússon. Sorting operators and their preimages. *Computer Science*, 2013. (Cited on pages 8, 11, 21, and 26.)

[10] HENNING Ú'LFARSSON. Ultra-quick tutorial for generalized pattern macros. *http://staff.ru.is/henningu/notes/patternmacros/patternmacros.tex*. (Cited on page 9.)