

SOURCE CODE 1. Generator: Permutationen in der “Gray–Code–artigen Aufzählung von Johnson und Trotter.

```
def generate_johnson_trotter(n):
    """Generator: Erzeuge die Permutationen von {1,2,...,n}
    mit dem Johnson-Trotter-Algorithmus."""

    # Initialisierung:
    # - Permutation pi (als numpy-array),
    # - inverse Permutation pi_inv(ebenso als numpy-array),
    # - und "Bewegungsrichtungen" der "wandernden Elemente"
    # sind dargestellt als numpy-arrays der Länge n+2.
    # Indices 0 und n+1 dienen nur der "Begrenzung"; das
    # eigentliche Permutationswort ist im "slice 1:n+1")
    # gespeichert.
    # Die numpy-Funktion arange(n) liefert Werte von 0 bis
    # n-1 in einem numpy-array:
    pi = np.arange(n+2, dtype=int)
    # Index 0 enthält - ebenso wie n+1 - eine "Begrenzung",
    # die während des Algorithmus nicht verändert wird:
    pi[0] = n+1
    pi_inv = np.arange(n+2, dtype=int)
    # Die numpy-Funktion ones(n) liefert ein numpy-array
    # der Länge n, dessen Eintragungen alle gleich 1 sind.
    # numpy-Vektoren kann man mit Skalaren (hier: -1) multiplizieren
    # und addieren - wie sich das für Vektoren gehört; -)
    dirs = (-1)*np.ones(n+2, dtype=int)
    # Datentyp dtype=int für die Einträge ist hier WICHTIG, da
    # diese teils als INDIZES in arrays verwendet werden!

    # Aktive Elemente (das sind die, die durch das Permutationswort
    # "wandern"): Anfangs alle außer 1.
    active = [x for x in range(2,n+1)]

    # Solange es aktive Elemente gibt, durchlaufen wir die
    # while-Schleife: len(active) ist zwar genau besehen eine
    # int-Zahl und kein bool-Wahrheitswert, wird aber "automatisch"
    # als Wahrheitswert interpretiert: False genau dann, wenn 0,
    # sonst True.
    while len(active):
        # Ausgabe des aktuellen Permutationswortes
        # (das, wie gesagt, dem "slice 1:-1" entspricht):
        yield ' '.join([str(x) for x in pi[1:-1]])
        # print(' '.join([str(x) for x in pi[1:-1]]))

        # Das größte der aktiven Elemente soll "wandern" ...
        m = max(active)
        # ... und zwar in die "entsprechende Richtung":

        # j ist die Stelle, an der Element m aktuell steht ...
        j = pi_inv[m]
```

```

# ... wir vertauschen das Element an Stelle j (also m)
# mit seinem "Nachbarn" (in Richtung dirs[m]) ...
neighbour_j = j + dirs[m]
pi[j] = pi[ neighbour_j ]
pi[ neighbour_j ] = m
# (Diese Vertauschung entspricht einer Transposition tau!)
# ... und passen die inverse Permutation entsprechend
# an, damit wieder pi[pi_inv[j]] = j für j=1,2,...n gilt:
pi_inv[m] = neighbour_j
pi_inv[pi[j]] = j

# Haben wir einen "Umkehrpunkt am Rand" erreicht
# Das ist dann der Fall, wenn wir - in dieselbe Richtung
# wie bisher weiter wandernd - auf ein Element > m stoßen
# würden (zur Erinnerung: pi[0] = pi[n+1] = n+1): Dann drehen
# wir die Richtung von m um und "deaktivieren" m.
if m < pi[j+2*dirs[m]]:
    dirs[m] = -dirs[m]
    active.remove(m)

# Nach dem "Schritt von m" lassen wir auch alle
# Elemente größer als m (wenn es welche gibt) wieder
# aktiv werden:
active+= [x for x in range(m+1,n+1)]

# Ausgabe des aktuellen Permutationswortes:
yield ' '.join([str(x) for x in pi[1:-1]])

```