

# Speeding Up Steinhaus-Johnson-Trotter Algorithm for Permutations

Seenu Reddi  
ReddiSS at aol dot com  
December 1, 2009

The Steinhaus-Johnson-Trotter (SJT) algorithm uses ingenious techniques for enumerating all permutations (see [wikipedia.org](http://wikipedia.org) or [cut-the-knot.org](http://cut-the-knot.org)). The permutations are generated in an order where the adjacent permutations differ only in one exchange of elements and this exchange consists of adjacent elements in the permutation. As an example, the algorithm generates the permutations for four elements 1, 2, 3 and 4 in the following sequence:

```
1 2 3 4 -> shift 4 left 4 times
1 2 4 3 -> (4, 3)
1 4 2 3 -> (3, 2), (1, 2)
4 1 2 3 -> apply sjt swap (3, 2)

4 1 3 2 -> shift 4 right 4 times
1 4 3 2
1 3 4 2
1 3 2 4 -> apply sjt swap (1, 0)

3 1 2 4 -> shift 4 left 4 times
3 1 4 2
3 4 1 2
4 3 1 2 -> apply sjt swap (3, 2)

4 3 2 1 -> shift 4 right 4 times
3 4 2 1
3 2 4 1
3 2 1 4 -> apply sjt swap (0, 1)

2 3 1 4 -> shift 4 left 4 times
2 3 4 1
2 4 3 1
4 2 3 1 -> apply sjt swap (2, 3)

4 2 1 3 -> shift 4 right 4 times
2 4 1 3
2 1 4 3
2 1 3 4
```

As can be seen the permutations generated differ only in exchange of adjacent elements and this has been pointed out in the literature as an advantage when computing tours in the Traveling Salesman Problem (TSP). Most of the C (C++) programs presented seem complex and we present a simple program to accomplish the SJT algorithm. We make the simplification in that we do not generate SJT swaps when we are shifting the largest element left and right.

```

// Generates swaps in S-J-T algorithm
// generates permutations applying these swaps

#include "stdio.h"

#define LEFT 0
#define RIGHT 1

int v[16], d[16];

void swap(int i, int j)
{
    int tmp;

    tmp = v[i];
    v[i] = v[j];
    v[j] = tmp;

    tmp = d[i];
    d[i] = d[j];
    d[j] = tmp;
}

void print_p(int n)
{
    int i;

    for(i = 0; i < n; i++)
        printf("%d ", v[i]);
    printf("\n");
}

int sjtSwap(int n, int start)
{
    int i, j;
    int k = -1, dir, s;
    static int sjtCount = 0;

    for (i = start; i < n + start; i++) {
        if (d[i] == LEFT && i > start + 0) {
            if (v[i] > v[i - 1]) {
                if (v[i] > k) {
                    k = v[i];
                    s = i;
                    dir = LEFT;
                }
            }
        } else if (d[i] == RIGHT && i < start + n - 1) {
            if (v[i] > v[i + 1]) {
                if (v[i] > k) {
                    k = v[i];
                    s = i;
                    dir = RIGHT;
                }
            }
        }
    }

    if (k == -1)
        return k;

    if (dir == LEFT) {
        swap(s, s - 1);
    } else {
        swap(s, s + 1);
    }

    // printf("%3d ", s * (dir ? 1 : -1));
    sjtCount++;
    if (sjtCount == 2 * n) {

```

```

    // printf("\n");
    sjtCount = 0;
}

for (i = start; i < start + n; i++) {
    if (v[i] <= k)
        continue;
    else
        d[i] = (d[i] + 1) % 2; // change direction
}
return k;
}

// S-J-T algorithm for n = 4
// 1 2 3 4 -> shift 4 left 4 times
// 1 2 4 3
// 1 4 2 3
// 4 1 2 3 -> apply sjt swap -3 (3, 2)

// 4 1 3 2 -> shift 4 right 4 times
// 1 4 3 2
// 1 3 4 2
// 1 3 2 4 -> apply sjt swap -1 (1, 0)

// 3 1 2 4 -> shift 4 left 4 times
// 3 1 4 2
// 3 4 1 2
// 4 3 1 2 -> apply sjt swap -3 (3, 2)

// 4 3 2 1 -> shift 4 right 4 times
// 3 4 2 1
// 3 2 4 1
// 3 2 1 4 -> apply sjt swap 0 (0, 1)

// 2 3 1 4 -> shift 4 left 4 times
// 2 3 4 1
// 2 4 3 1
// 4 2 3 1 -> apply sjt swap 2 (2, 3)

// 4 2 1 3 -> shift 4 right 4 times
// 2 4 1 3
// 2 1 4 3
// 2 1 3 4

void main(int argc, char *argv[])
{
    int n = atoi(argv[1]);
    int i, cnt, k;

    for (i = 0; i < n; i++) {
        v[i] = i + 1;
        d[i] = LEFT;
    }

    print_p(n);
    cnt = 1;
    while (1) {
        // generate n permutations travelling left with n moving
        for (i = n - 1; i > 0; i--) {
            swap(i, i - 1);
            cnt++;
            print_p(n);
        }

        // change direction
        k = sjtSwap(n - 1, 1);
        if (k == -1) {
            printf("\nperms = %d\n", cnt);
            return;
        }
        cnt++;
    }
}

```

```

print_p(n);

// generate n permutations travelling right with n moving
for (i = 0; i < n - 1; i++) {
    swap(i, i + 1);
    print_p(n);
    cnt++;
}

// change direction
k = sjtSwap(n - 1, 0);
if (k == -1) {
    printf("\nperms = %d\n", cnt);
    return;
}
cnt++;
print_p(n);
}
}

```

Programs are available as rared programs at [www.rspq.org/pubs](http://www.rspq.org/pubs). For speeding up the process of generating the permutations, we look at the swaps generated by the algorithm and in our example for four elements we note (not counting the left and right shift swaps of the largest element):

-3, -1, -3, 0, 2

where we use the notation  $-x$  for the swap  $(x, x - 1)$  and  $x$  for the swap  $(x, x + 1)$ . (Note swap 0 can only mean  $(0, 1)$  and thus we do not have to deal with  $+0$  or  $-0$ ). We try to detect any patterns and peculiarities present in these swaps and use them to speed up the process of generating the swaps and hence the permutations. For six and eight elements, the following swaps are observed:

S-J-T Swaps for n = 6

```

-5 -3 -3 -1 -5 0 2 2 4 -2
-5 -3 -3 -1 -3 0 2 2 4 -3
-5 -3 -3 -1 2 0 2 2 4 1
-5 -3 -3 -1 4 0 2 2 4 -1
-5 -3 -3 -1 -5 0 2 2 4 -2
-5 -3 -3 -1 -3 0 2 2 4 -3
-5 -3 -3 -1 2 0 2 2 4 1
-5 -3 -3 -1 4 0 2 2 4 0
-5 -3 -3 -1 -5 0 2 2 4 -2
-5 -3 -3 -1 -3 0 2 2 4 2
-5 -3 -3 -1 2 0 2 2 4 1
-5 -3 -3 -1 4 0 2 2 4

```

S-J-T Swaps for n = 8

```

-7 -5 -5 -3 -3 -1 -7 0 2 2 4 4 6 -4
-7 -5 -5 -3 -3 -1 -5 0 2 2 4 4 6 -2
-7 -5 -5 -3 -3 -1 -3 0 2 2 4 4 6 -5
-7 -5 -5 -3 -3 -1 2 0 2 2 4 4 6 1
-7 -5 -5 -3 -3 -1 4 0 2 2 4 4 6 3
-7 -5 -5 -3 -3 -1 6 0 2 2 4 4 6 -3
-7 -5 -5 -3 -3 -1 -7 0 2 2 4 4 6 -4
-7 -5 -5 -3 -3 -1 -5 0 2 2 4 4 6 -2
-7 -5 -5 -3 -3 -1 -3 0 2 2 4 4 6 -3
-7 -5 -5 -3 -3 -1 2 0 2 2 4 4 6 1
-7 -5 -5 -3 -3 -1 4 0 2 2 4 4 6 3
-7 -5 -5 -3 -3 -1 6 0 2 2 4 4 6 -1
-7 -5 -5 -3 -3 -1 -7 0 2 2 4 4 6 -4
-7 -5 -5 -3 -3 -1 -5 0 2 2 4 4 6 -2
-7 -5 -5 -3 -3 -1 -3 0 2 2 4 4 6 -5

```











```

-5 -3 -3 -1 -5 0 2 2 4 -2
-5 -3 -3 -1 -3 0 2 2 4 -3
-5 -3 -3 -1 2 0 2 2 4 1
-5 -3 -3 -1 4 0 2 2 4 -1
-5 -3 -3 -1 -5 0 2 2 4 -2
-5 -3 -3 -1 -3 0 2 2 4 -3
-5 -3 -3 -1 2 0 2 2 4 1
-5 -3 -3 -1 4 0 2 2 4 0
-5 -3 -3 -1 -5 0 2 2 4 -2
-5 -3 -3 -1 -3 0 2 2 4 2
-5 -3 -3 -1 2 0 2 2 4 1
-5 -3 -3 -1 4 0 2 2 4

```

we can store  $\{-5, -3, -1, X, 0, 2, \dots, Y\}$  as a template. We can start generating the swaps by reading off  $-5, -3, ..$  and when we come to  $X$ , we can use the template  $\{-5, -3, 2, 4\}$ . We can adopt a similar procedure for  $Y$  though the template for  $Y$  may not be as simple as for  $X$  since it does not have the periodicity of 4. When the number of elements is odd, one sees similar patterns but slightly more complicated. We do not concern ourselves with this added complexity since we can generate permutations for the even number of permutations and from these, one may generate for the next odd number.

```

int S8N = 8;
int S8Count = 40320;
int S8TotalColumns = 14;
int S8MidColumn = 6; // starts from 0
int S8EndColumn = 13;

int S8EndColMidMarker = 2;
int S8EndColEndMarker = 5;

char S8Swaps[] = { -7,  -5,  -5,  -3,  -3,  -1,  -7,   0,   2,   2,   4,   4,   6,  -4};
int S8MidColCount = 6;
char S8MidCol[] = { -7,  -5,  -3,   2,   4,   6};

int S8EndColCount = 6;
char S8EndCol[] = {-4,  -2,  -5,   1,   3,  -3};

char S8EndColMid[] = {
-5,  -3,  -5,   2,   4,
-5,  -3,  -3,   2,   4,
-5,  -3,   2,   2,   4,
-5,  -3,   4,   2,   4,
-5,  -3,  -5,   2,   4,
-5,  -3,  -3,   2,   4,
-5,  -3,   2,   2,   4,
-5,  -3,   4,   2,   4,
-5,  -3,  -5,   2,   4,
-5,  -3,  -3,   2,   4,
-5,  -3,   2,   2,   4,
-5,  -3,   4,   2,   4
};

char S8EndColEnd[] = {
-3,  -1,   0,   2,  -2,
-3,  -1,   0,   2,  -3,
-3,  -1,   0,   2,   1,
-3,  -1,   0,   2,  -1,
-3,  -1,   0,   2,  -2,
-3,  -1,   0,   2,  -3,
-3,  -1,   0,   2,   1,

```

```

-3, -1, 0, 2, 0,
-3, -1, 0, 2, -2,
-3, -1, 0, 2, 2,
-3, -1, 0, 2, 1,
-3, -1, 0, 2
};

int S6N = 6;
int S6Count = 720;
int S6TotalColumns = 10;
int S6MidColumn = 4; // starts from 0
int S6EndColumn = 9;

int S6EndColMidMarker = 1;
int S6EndColEndMarker = 3;

char S6Swaps[] = { -5, -3, -3, -1, -5, 0, 2, 2, 4, -2};
int S6MidColCount = 4;
char S6MidCol[] = {-5, -3, 2, 4};

int S6EndColCount = 4;
char S6EndCol[] = { -2, -3, 1, -1};

char S6EndColMid[] = {
-3, -3, 2
};

char S6EndColEnd[] = {
-1, 0
};

struct _Parameters {
    int N;
    int Count;
    int TotalColumns;
    int MidColumn;
    int EndColumn;
    int MidColCount;
    int EndColCount;
    int EndColMidMarker;
    int EndColEndMarker;

    char *Swaps;
    char *MidCol;
    char *EndCol;
    char *EndColMid;
    char *EndColEnd;
};

typedef struct _Parameters PARAMETERS;
PARAMETERS S6, S8, S10, S12;

#define ASSIGN(S) \
    S###N = S##N;\
    S###Count = S##Count;\
    S###TotalColumns = S##TotalColumns;\
    S###MidColumn = S##MidColumn;\
    S###EndColumn = S##EndColumn;\
    S###MidColCount = S##MidColCount;\
    S###EndColCount = S##EndColCount;\
    S###EndColMidMarker = S##EndColMidMarker;\
    S###EndColEndMarker = S##EndColEndMarker;\
    S###Swaps = S##Swaps;\
    S###MidCol = S##MidCol;\
    S###EndCol = S##EndCol;\
    S###EndColMid = S##EndColMid;\
    S###EndColEnd = S##EndColEnd

#define SETVARS(S) \
    N = S###N;\
    Count = S###Count;\

```

```

TotalColumns = S##.##TotalColumns;\
MidColumn = S##.##MidColumn;\
EndColumn = S##.##EndColumn;\
MidColCount = S##.##MidColCount;\
EndColCount = S##.##EndColCount;\
EndColMidMarker = S##.##EndColMidMarker;\
EndColEndMarker = S##.##EndColEndMarker;\
Swaps = S##.##Swaps;\
MidCol = S##.##MidCol;\
EndCol = S##.##EndCol;\
EndColMid = S##.##EndColMid;\
EndColEnd = S##.##EndColEnd

int Permutation[16];

void print_p(int *p, int n)
{
    int i;

    for (i = 0; i < n; i++) {
        printf("%3d", p[i]);
        if (i != n - 1)
            printf(", ");
    }
    printf("\n");
}

char GetInversion(int index, PARAMETERS S)
{
    int i;
    static int im, imc, iec, iecm, iece;

    static int N;
    static int Count;
    static int TotalColumns;
    static int MidColumn;
    static int EndColumn;
    static int MidColCount;
    static int EndColCount;
    static int EndColMidMarker;
    static int EndColEndMarker;

    static char *Swaps;
    static char *MidCol;
    static char *EndCol;
    static char *EndColMid;
    static char *EndColEnd;

    char cycle;

    if (index == 0) {
        SETVARS(S);
        imc = iec = iecm = iece = 0;
    }

    i = index;
    im = i % TotalColumns;
    if (im == MidColumn) {
        cycle = MidCol[imc++];
        imc = imc % MidColCount;
    } else if (im == EndColumn) {
        if (iec == EndColMidMarker) {
            cycle = EndColMid[iecm++];
        } else if (iec == EndColEndMarker) {
            cycle = EndColEnd[iece++];
        } else {
            cycle = EndCol[iec];
        }
        iec++;
        iec = iec % EndColCount;
    } else {

```

```

    cycle = Swaps[im];
}
return cycle;
}

#define swap(i1, i2) {__i = Permutation[i1]; Permutation[i1] = Permutation[i2];
Permutation[i2] = __i;}
void GenPerm(PARAMETERS S)
{
    int i, cnt, Count, n, ix, __i;
    char cycle, p1, p2;

    n = S.N;
    Count = S.Count;
    for (i = 0; i < n; i++)
        Permutation[i] = i + 1;

    ix = 0;
    cnt = 1;
    print_p(Permutation, n);
    while (1) {
        for (i = n - 1; i > 0; i--) {
            swap(i, i - 1);
            cnt++;
            print_p(Permutation, n);
        }
        if (cnt == Count) {
            break;
        }

        cycle = GetInversion(ix++, S);
        if (cycle >= 0) {
            p1 = cycle;
            p2 = p1 + 1;
        } else {
            p1 = -cycle;
            p2 = p1 - 1;
        }
        swap(p1, p2);
        print_p(Permutation, n);
        cnt++;
        if (cnt == Count) {
            break;
        }

        // generate n permutations travelling right with n moving
        for (i = 0; i < n - 1; i++) {
            swap(i, i + 1);
            cnt++;
            print_p(Permutation, n);
        }
        if (cnt == Count) {
            break;
        }
    }

    cycle = GetInversion(ix++, S);
    if (cycle >= 0) {
        p1 = cycle;
        p2 = p1 + 1;
    } else {
        p1 = -cycle;
        p2 = p1 - 1;
    }
    swap(p1, p2);
    print_p(Permutation, n);
    cnt++;
    if (cnt == Count) {
        break;
    }
}
}

```

```
void main(int argc, char *argv[])
{
    if (atoi(argv[1]) == 6) {
        ASSIGN(S6);
        GenPerm(S6);
    } else if (atoi(argv[1]) == 8) {
        ASSIGN(S8);
        GenPerm(S8);
    } else if (atoi(argv[1]) == 10) {
        ASSIGN(S10);
        GenPerm(S10);
    } else if (atoi(argv[1]) == 12) {
        ASSIGN(S12);
        GenPerm(S12);
    }
}
```

A complete C program for elements from 6 to 12 is contained in the rared file sjt.rar available at [www.rspq.org/pubs](http://www.rspq.org/pubs). Though we do not see appreciable speed improvements up to 10 elements, the proposed approach is at least two times faster than the conventional S-J-T algorithm and appears to be four times faster than the lexicographic and non-lexicographic generators.

There are many interesting open questions as to how to characterize the swap sequences and whether there exist other procedures capable of generating all the permutations like S-J-T algorithm.