Permutation Generation: Algorithms 5-9

Samuel Grant Dawson Williams. (ID: 93897111 User: sgw35)

May 17, 2007

Contents

Listings

1 Introduction

Permutation algorithms are one element of the earth from which all fields of computer science can grow. Like a sturdy tree, a programmer must have strong roots in this fundamental field.

Here, I present 5 different permutation algorithms, from the worst, running in $O(n^n)$, to the current best running in $O(n!)$, with worst case $O(1)$.

However, algorithmic cost and implementation speed can be very different, depending on what hardware we are using, so I will also test, analyse and optimise the performance of these algorithms running on real computers.

2 Algorithm 5; Slow Recursive

Listing 1: Algorithm 5; Slow Recursive

```
class Algorithm5 : public AlgorithmBase {
protected:
    const int n;
    vector<bool> used;
    array_t a;

    void perm(unsigned k) {
        if (k <= n) {
            for (unsigned i = 1; i <= n; ++i) {
                if (!used[i]) {
                    a[k] = i;
                    used[i] = true;
                    perm(k+1);
                    used[i] = false;
                }
            }
        } else {
            addResult(a);
        }
    }
public:
    Algorithm5 (int _n) : n(_n) {
        a.resize(n+1);
        used.resize(n+1);
    }

    virtual void generate () {
        resetResults();
        perm(1);
    }
};
```

This recursive algorithm generates permutations in lexicographic order. $perm(k)$ will generate all values $1..n$ and will put this in $a[k]$ when this value has not been *used* already. Initially, no values are *used*, so the used array is all false. Once we have placed all elements into a, then we have a correct unique permutation - the *used* array will be all true in this case.

Because i goes from $1..n$ the algorithm will produce lexicographic order. This algorithm will be slow because the inner loop considers almost all possibilities. For example, when $size = 4$, we will generate permutations as seen in table 1.

output	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$used[0]$	$used[1]$	$used[2]$	$used[3]$	status
	1				t	f	f	f	continue
	1	1			t	f	f	f	return
	1	2			t	t	f	f	continue
	1	2	1		t	t	f	f	return
	1	2	2		t	t	f	f	return
	1	2	3		t	t	t	f	continue
	1	2	3	1	t	t	t	f	return
	1	2	3	2	t	t	t	f	return
	1	2	3	3	t	t	t	f	return
$*$	1	2	3	4	t	t	t	t	okay
	1	2	4	1	t	t	f	t	return
	1	2	4	2	t	t	f	t	return
$*$	1	2	4	3	t	t	t	t	okay
	1	2	4	4	t	t	f	t	return
	1	3	1		t	f	t	f	return
	1	3	2		t	t	t	f	continue
	1	3	2	1	t	t	t	f	return
	1	3	2	2	t	t	t	f	return
	1	3	2	3	t	t	t	f	return
$*$	1	3	2	4	t	t	t	t	okay

Table 1: Algorithm 5 is very inefficient, as it explores almost all possibilities. The star under output indicates that we have the correct permutation.

3 Algorithm 6; Faster Recursive

Listing 2: Algorithm 6; Faster Recursive

```
class Algorithm6 : public AlgorithmBase {
protected:
    vector<array_t> results;
    const int n;
    array_t a;

    unsigned minimum (unsigned k) {
        unsigned j = 0, t = (unsigned)-1;

        for (unsigned i = k; i <= n; ++i) {
            if (a[i] < t && a[i] > a[k-1]) {
                j = i;
                t = a[i];
            }
        }

        return j;
    }

    void swap (unsigned i, unsigned j) {
        unsigned t = a.at(i);
        a[i] = a.at(j);
        a[j] = t;
    }

    void reverse (unsigned k) {
        for (unsigned i = 0; i < (n-k+1) / 2; i += 1)
            swap(k+i, n-i);
    }

private:
    void perm(unsigned k) {
        int j;

        if (k < n) {
            for (unsigned i = k; i <= n; ++i) {
                perm(k+1);

                if (i != n) {
                    reverse(k+1);
                    j = minimum(k+1);
                    swap(k, j);
                    addResult(a);
                }
            }
        }
    }
public:
    Algorithm6 (int _n) : n(_n) {
        a.resize(n+1);
    }

    virtual void generate () {
        resetResults();

        for (unsigned i = 0; i <= n; ++i)
            a[i] = i;

        addResult(a);
        perm(1);
    }
};
```

This recursive algorithm generates permutations in lexicographic order. The set of all available items is initialised in order, and maintained in order; After the call to $perm(k + 1)$, the available set of items is in decreasing order, so we call $reverse(k + 1)$ to put it back in order.

If we consider any $perm(k)$, because of the above property, values after k will be ascending. Once we have found all permutations for the subset $a[k + 1...n]$, that portion will be in reverse order.

To find the next permutation, we will find the minimum value in this set greater than the current value at k, and swap it. Then, we descend again to generate the subset of permutations, with a new value.

In the simplest case, this will swap the last two elements. Because we always swap the minimum value from the subset with $a[k]$, we will produce lexicographic order (i.e. from minimum value to maximum value).

For example, for the permutation [1 2 4 3] generated in $perm(3)$ will return back to $perm(2)$, which will then reverse [4 3] to form [1 2 3 4]. Then, the minimum value greater than $a[k]$ in [3 4] will be swapped. Thus, the result is [1 3 2 4], the next lexicographic permutation. For a detailed example, see table 2.

output	action	$a[0]$	$a[1]$	$a[2]$	$a[3]$	k
$*$	$perm(1)$	1	[2	3	4]	1
	$perm(2)$	1	2	[3	4]	2
	$perm(3)$	1	2	3	[4]	3
	$reverse(3+1)$	1	2	3	[4]	3
$*$	$swap(3, min(3+1))$	1	2	4	[3]	3
	$reverse(2+1)$	1	2	[3	4]	2
$*$	$swap(2, min(2+1))$	1	3	[2	4]	2

Table 2: The brackets in a represent the portion $a[k+1]..a[n]$ and we can call this the available set. The star under output indicates that we have the correct permutation.

4 Algorithm 7; Iterative

Listing 3: Algorithm 7; Iterative

```
class Algorithm7 : public Algorithm6 {
public:
    Algorithm7 (int _n) : Algorithm6(_n) {

    }

    virtual void generate () {
        resetResults();
        unsigned i;

        for (i = 0; i <= n; ++i) a[i] = i;

        addResult(a);
        do {
            i = n;

            while (a[i-1] > a[i]) i -= 1;

            i -= 1;
            if (i > 0) {
                reverse(i+1);
                unsigned j = minimum(i+1);
                swap(i, j);
                addResult(a);
            }
        } while (i > 0);
    }
};
```

This algorithm functions in a similar method to algorithm 6, but instead of using recursion to keep track of an available set, we consider the biggest ascending portion on the right hand side. To find this subset, we scan from the right hand side towards the left, until we find a value less than the current. We can call this a hill climbing technique (see figure 2).

As we know that this portion is the biggest possible permutation for this particular subset, we can make no bigger permutation, and our only choice is to swap in a new value from the left hand side. Firstly, we reverse this subset, so we have the start of a new subset. Then we swap the value left of this subset, $a[i]$, with the minimum value greater than $a[i]$ in this subset.

For example, in the permutation [1 4 3 2] the biggest ascending portion from the right is [4 3 2]. We will reverse this portion to form [1 2 3 4] and substitute values 1 and 2 to form the next permutation [2 1 3 4] (see table 3 for a bigger example).

output	action	$a[0]$	$a[1]$	$a[2]$	$a[3]$	i
$*$	i = 3	1	2	3	[4]	3
	reverse(i+1)	1	2	3	[4]	3
$*$	swap(i, min(i+1))	1	2	4	[3]	3
	i = 2	1	2	[4	3]	2
	reverse(i+1)	1	2	[3	4]	2
$*$	swap(i, min(i+1))	1	3	[2	4]	2
	i = 3	1	3	2	[4]	3
	reverse(i+1)	1	3	2	[4]	3
$*$	swap(i, min(i+1))	1	3	4	[2]	3

Table 3: The brackets in a represent the portion $a[i+1]..a[n]$. This represents the largest ascending portion from the right hand side. The star under output indicates that we have the correct permutation.

5 Algorithm 8: Johnson-Trotter Recursive

Listing 4: Algorithm 8; Johnson-Trotter Recursive

```
class Algorithm8 : public AlgorithmBase {
protected:
    const int n;
    array_t a, d, p;

    void move (int x) {
        int w = a[p[x]+d[x]];
        a[p[x]+d[x]] = x;
        a[p[x]] = w;
        p[w] = p[x];
        p[x] = p[x]+d[x];
    }

private:
    void nest(int k) {
        if (k <= n) {
            for (int i = 1; i <= k; i += 1) {
                nest(k+1);
                if (i < k) {
                    move(k);
                    addResult(a);
                }
            }

            d[k] = -d[k];
        }
    }

public:
    Algorithm8 (int _n) : n(_n) {
        a.resize(n+1);
        d.resize(n+1);
        p.resize(n+1);
    }

    virtual void generate () {
        resetResults();

        for (int i = 1; i <= n; i += 1) {
            a[i] = i;
            p[i] = i;
            d[i] = -1;
        }

        addResult(a);
        nest(2);
    }
};
```

This recursive algorithm works by interleaving individual values within the overall permutation. The fastest moving value is the right-most, while the slowest moving value is the 2nd to left (the left most value is never the target to *move* - but it does get swapped by other values moving into its place). Because the value can move from left to right, and from right to left, we must keep track of its direction in $d[k]$.

The parameter to *move* is not the position to move, it is the value, and this is done through using the p array which contains the location of each individual value.

The operation *move*(k) swaps element at $p[k]$ in with the one in direction $d[k]$ (either $+1$ or -1), and updates p. p is used to hold the current location of a particular value within a, this is needed because we will move a particular value k in *nest*(k). d is reversed at the end of each set of children, as we will then move this value back through a in the opposite direction.

Table 4 steps through the first few permutations to show the interleaving of values 4 and 3. Figure 1 shows the structure of calls to *move*.

output	action	$a[0]$	$a[1]$	$a[2]$	$a[3]$	k	i	p
$*$	nest(3)	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	2	1	[1 2 3 4]
	nest(4)	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	$\leftarrow 4$	3	1	[1 2 3 4]
$*$	move(4)	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 4$	$\leftarrow 3$	4	1	[1 2 4 3]
$*$	move(4)	$\leftarrow 1$	$\leftarrow 4$	$\leftarrow 2$	$\leftarrow 3$	4	2	[1 3 4 2]
$*$	move(4)	$\leftarrow 4$	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	4	3	[2 3 4 1]
	d[k] = -d[k]	$\rightarrow 4$	$\leftarrow 1$	$\leftarrow 2$	$\leftarrow 3$	4	4	[2 3 4 1]
$*$	move(3)	$\rightarrow 4$	$\leftarrow 1$	$\leftarrow 3$	$\leftarrow 2$	3	2	[2 4 3 1]
$*$	move(4)	$\leftarrow 1$	$\rightarrow 4$	$\leftarrow 3$	$\leftarrow 2$	4	1	[1 4 3 2]
$*$	move(4)	$\leftarrow 1$	$\leftarrow 3$	$\rightarrow 4$	$\leftarrow 2$	4	1	[1 4 2 3]

Table 4: This is a simplified look at the internals of Algorithm 8. Some steps are expanded so the function can be understood more easily. The star under output indicates that we have the correct permutation. The arrow indicates the direction the value is moving in.

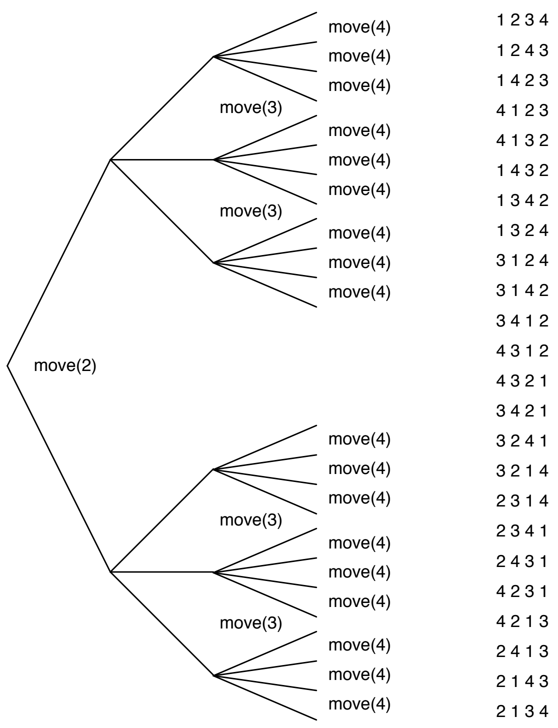

Figure 1: The structure of *move*(x) calls within the scope of a tree.

6 Algorithm 9: Johnson-Trotter Iterative

Listing 5: Algorithm 9; Johnson-Trotter Iterative

```
class Algorithm9 : public Algorithm8 {
protected:
    array_t c, up;

public:
    Algorithm9 (int _n) : Algorithm8(_n) {
        c.resize(n+1);
        up.resize(n+1);
    }

    virtual void generate () {
        resetResults();
        int i;

        for (i = 1; i <= n; i += 1) {
            a[i] = i;
            c[i] = 0;
            d[i] = -1;
            p[i] = i;
            up[i] = i;
        }

        addResult(a);

        do {
            i = up[n];
            up[n] = n;

            if (i > 1) {
                c[i] = c[i] + 1;
                move(i);
                addResult(a);
            }

            if (c[i] == i-1) {
                up[i] = up[i-1];
                up[i-1] = i - 1;
                d[i] = -d[i];
                c[i] = 0;
            }
        } while (i != 1);
    }
};
```

We analyse the structure of the recursive algorithm 8, and serialise this series of calls into an iterative function. We use two new arrays *up* and *c* to keep track of the location of the current position in the tree. You can imagine that *up* array controls our position from the base of the tree to the top of the tree, and *c* keeps track of how many times we need to move at the current level, i.e. the position from one side of the tree to the other.

We can simply describe the set of values to move when $size = 4$ as an example:

```
0: n, n, n,           (c[n]   => 3 times)
1:         n-1,       (c[n-1] => 2 times)
0: n, n, n,
1:         n-1,
0: n, n, n,
2:             n-2,   (c[n-2] => 1 time)
0: n, n, n,
1:         n-1,
0: n, n, n,
1:         n-1,
0: n, n, n.
```

We can see that this sequence represents a modified form of the binary carry sequence [0 1 0 2 0 1 0]. The array *up* alone creates this sequence, and c augments it so that the tree has j+1 children at height j. This algorithm for *up* can be simplified to listing 6. We can consider every node a last child.

Listing 6: Binary Carry Sequence in O(1) worst case

```
n = 4; up = (0...n).to_a

def output(x)
  puts "move(#{x})"
end

begin
  i = up[n-1]
  up[n-1] = n-1

  output(n - 1 - i) if (i > 0)

  up[i] = up[i-1]
  up[i-1] = i - 1
end until i == 0
```

Program listing 6 will output the sequence [0 1 0 2 0 1 0].

We use c to count how many times we have to *move*(i) at a particular level. When we use c, we also consider the position of the value in c - this is the maximum number of times we can execute this move. For example, at $c[n]$, we can execute *move*(n) $n-1$ times and $c[n-1]$ we can execute *move*$(n-1)$ $n-2$ times, before we come to the last child condition.

There are two procedures in the main recursive function. The first part is executed almost every time:

```
            if (i > 1) {
                c[i] = c[i] + 1;
                move(i);
                addResult(a);
            }
```

This code replicates the behaviour of the recursive function. Here, i is essentially equivalent to k. i is always equal to $up[n]$.

The second part we call the last child condition:

```
            if (c[i] == i-1) {
                up[i] = up[i-1];
                up[i-1] = i - 1;
                d[i] = -d[i];
                c[i] = 0;
            }
```

This simulates moving back up the the tree by one level. We copy $up[i-1]$ into $up[i]$. This eventually propagates into i itself. We also set $up[i-1]$ to $i-1$ to maintain tree's order. Because this is the last child, the count at this level is reset to zero $c[i] = 0$.

7 Sorter and Tester

Listing 7: Sorter and Tester

```
void quicksort(array_t &v, unsigned left, unsigned right) {
    if (right > left + 1) {
        int pVal = v[left];
        int l = left + 1, r = right;

        while (l < r) {
            if (v[l] < pVal)
                l += 1;
            else
                std::swap(v[l], v[--r]);
        }

        swap(v[--l], v[left]);
        quicksort(v, left, l);
        quicksort(v, r, right);
    }
}

void sort (array_t &v) {
    quicksort(v, 0, v.size());
}

bool sorted (array_t &v) {
    int prev = v[0];

    for (unsigned i = 1; i < v.size(); i += 1) {
        if (v[i] < prev) return false;

        prev = v[i];
    }

    return true;
}

array_t permutate(array_t &p, array_t &d) {
    array_t r; r.reserve(d.size());

    for (unsigned i = 0; i < p.size(); i += 1)
        r.push_back(d[p[i]-1]);

    return r;
}

void test () {
    const int c = 4;
    Algorithm8 algo(c);
    algo->generate();

    vector<array_t> results (algo->getResults());

    array_t data; data.reserve(c);
    for (unsigned i = 0; i < c; i += 1) data.push_back(rand() % 1000);

    for (unsigned i = 0; i < results.size(); i += 1) {
        cout << "data␣␣␣␣"; printArray(data);
        cout << "permute␣"; printArray(results[i]);

        array_t p = permutate(results[i], data);
        cout << "permuted"; printArray(p);

        sort(p);
        cout << "sorted␣␣"; printArray(p);

        if (!sorted(p)) cout << "***␣Sort␣failed!!" << endl;

        if (i < results.size() - 1) cout << endl;
    }
}
```

Function *permutate* takes two arrays - one is a permutation, and the other a set of data, and produces a reordered set of data. Function *sorted* tests whether an array is in ascending order. Function *sort* sorts an input set of data using *quicksort* in ascending order.

Function *quicksort* is a divide and conquer sorting algorithm that organises data based on a pivot value. We pick a pivot value out of the list to be sorted, and reorder the list so that elements less than the pivot are on the left hand side, and elements greater than the pivot are on the right hand side. At this point, the pivot is in its final position. This step is often referred to as partitioning. We then apply this function recursively to the left and right sub-lists. We stop partitioning when the sub-list is of size 1. When all recursion has stopped, the list will be sorted completely.

The worst case of quicksort is $O(n^2)$, however in practice it is typically as efficient as $O(n \log n)$.

8 Performance

Algorithm	Total Time	Amortised Time	N = 11	N = 12	N = 13
5: Slow Recursive	$O(n^n)$	$O(e^n\sqrt{2\pi n})$	9.01	112.67	1514.9
6: Faster Recursive	$O(n!)$	$O(1)$	1.57	21.55	244.3
7: Iterative	$O(n!)$	$O(1)$	1.39	16.67	219.0
8: J-T Recursive	$O(n!)$	$O(1)$	0.93	11.41	154.3
9: J-T Iterative	$O(n!)$	$O(1)$	0.98	11.60	152.9
9: J-T Optimised	$O(n!)$	$O(1)$	0.81	9.50	124.3

Table 5: A summary of algorithm performance. Time is in seconds.

Algorithm 5 performance is terrible for any size greater than about 6. It is difficult for us to estimate the actual time of this algorithm, however its performance is bounded with $O(n^n)$, with $O(\frac{n^n}{n!})$ time per permutation. The amortised time can be bounded by $O(e^n\sqrt{2\pi n})$.

Algorithms 6 and 7, are much better. They have performance $O(n!)$, with amortised time $O(1)$. The iterative version performs slightly better.

Recursion is removed from algorithm 7, and in its place a hill climbing search (see figure 2) which also runs in amortised $O(1)$ time. In this case, the memory access overhead appears to be less than the cost of recursion, and thus it can run faster.

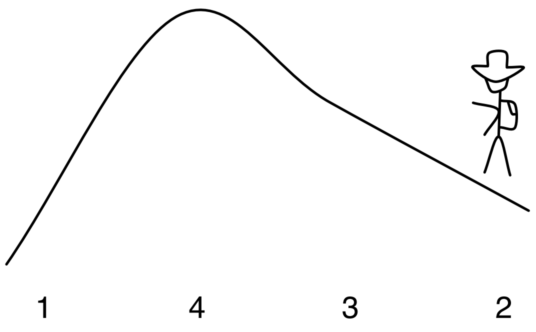

Figure 2: We must climb to the highest point.

Algorithms 8 and 9 are the most efficient both performing in $O(n!)$ time. Algorithm 8 has amortised time $O(1)$, while algorithm 9 has worst case time of$O(1)$.

The timing data suggests that algorithm 8 is the most efficient, with algorithm 9 almost the same, but slightly longer in most cases.

This suggests that with this implementation of algorithms 8 and 9, recursion is faster than the additional memory cost of an iterative implementation. The cost of function call is removed in the iterative implementation, but the cost of several additional memory accesses is added. Because there is more data involved, the computer CPU cache and instruction pipeline may be less effective, compounding the problem, where as recursion can be performed directly on the CPU, and in many cases is only 1 or 2 instructions.

On a platform where function call is more expensive, we would most likely get a different result.

If memory accesses of algorithm 9 could be reduced, it may be possible for it to become faster than algorithm 8.

9 Optimization

Listing 8: Algorithm 9; Optimised

```
class Algorithm9 : public Algorithm8 {
protected:
    array_t c, up;

    inline int sign(int x) {
        return x < 0 ? -1 : 1;
    }

    void move (int x) {
        int dx, px = p[x];
        if (px < 0) {
            dx = -1;
            px *= -1;
        } else {
            dx = 1;
        }

        int w = a[px+dx];
        a.at(px+dx) = x;
        a.at(px) = w;

        p.at(w) = px * sign(p[w]);
        p.at(x) = (px+dx) * dx;
    }

public:
    Algorithm9 (int _n) : Algorithm8(_n) {
        c.resize(n+1);
        up.resize(n+1);
    }

    virtual void generate () {
        resetResults();
        int i;

        for (i = 1; i <= n; i += 1) {
            a[i] = i;
            c[i] = 0;
            p[i] = -i;
            up[i] = i;
        }

        addResult(a);

        do {
            i = up[n];

            if (i == n) {
                for (unsigned w = 1; w < n; w += 1) {
                    move(i);
                    addResult(a);
                }

                up[n] = up[n - 1];
                up[n-1] = n - 1;
                p[n] = -p[n];

                continue;
            }

            up[n] = n;

            if (i > 1) {
                c[i] = c[i] + 1;
                move(i);
                addResult(a);
            }

            if (c[i] == i-1) {
                up[i] = up[i-1];
                up[i-1] = i - 1;
                p[i] = -p[i];
                c[i] = 0;
            }

        } while (i != 1);
    }
};
```

Two optimisations yeild approximately a 20% performance improvement. The first optimisation is to use array *p* to contain the direction *d*. This provides about 5-10% improvements in most cases. We trade memory access to *d* for additional integer manipulations on array *p*. The cost of a single memory access is very high compared to applying some integer operation to the CPU registers, so if we can swap a memory access for a few register manipulations, we can decrease the algorithms latency from step to step.

The second optimisation, which consistently yields approximately 10% increase in performance, is the replacement with the bottom loop with an inner loop. It can be seen that this algorithm spends half of its time at the bottom of the tree, where $i = n$. So making this into a single loop can avoid many needless assignments and comparisons:

```
            if (i == n) {
                for (unsigned w = 1; w < n; w += 1) {
                    move(i);
                    addResult(a);
                }

                up[n] = up[n - 1];
                up[n-1] = n - 1;
                p[n] = -p[n];

                continue;
            }
```

The timing characteristics of the algorithm are described in table 6. Both the optimisations are independent, so we can look at their impact separately, and together.

Because both of these optimisations in some form can be applied to algorithm 8 also, overall, algorithm 8 can probably still be faster, but more analysis and testing would be required to confirm this.

Inner Loop	*p* Move	-Os	Baseline %	-O3	Baseline %
No	No	0.493	0.0%	0.0755	0.0%
Yes	No	0.448	9.2%	0.0682	9.7%
No	Yes	0.446	9.6%	0.0718	5.0%
Yes	Yes	0.402	18.5%	0.0581	23.0%

Table 6: Performance comparison of various optimisations to Algorithm 9. Time in seconds. Percentage indicates time reduction.

10 Example output for N = 4

```
Generating permutations using algorithm 5: Slow Recursive order 4
     0:    1,    2,    3,    4
     1:    1,    2,    4,    3
     2:    1,    3,    2,    4
     3:    1,    3,    4,    2
     4:    1,    4,    2,    3
     5:    1,    4,    3,    2
     6:    2,    1,    3,    4
     7:    2,    1,    4,    3
     8:    2,    3,    1,    4
     9:    2,    3,    4,    1
    10:    2,    4,    1,    3
    11:    2,    4,    3,    1
    12:    3,    1,    2,    4
    13:    3,    1,    4,    2
    14:    3,    2,    1,    4
    15:    3,    2,    4,    1
    16:    3,    4,    1,    2
    17:    3,    4,    2,    1
    18:    4,    1,    2,    3
    19:    4,    1,    3,    2
    20:    4,    2,    1,    3
    21:    4,    2,    3,    1
    22:    4,    3,    1,    2
    23:    4,    3,    2,    1
```

```
Generating permutations using algorithm 6: Faster Recursive order 4
     0:    1,    2,    3,    4
     1:    1,    2,    4,    3
     2:    1,    3,    2,    4
     3:    1,    3,    4,    2
     4:    1,    4,    2,    3
     5:    1,    4,    3,    2
     6:    2,    1,    3,    4
     7:    2,    1,    4,    3
     8:    2,    3,    1,    4
     9:    2,    3,    4,    1
    10:    2,    4,    1,    3
    11:    2,    4,    3,    1
    12:    3,    1,    2,    4
    13:    3,    1,    4,    2
    14:    3,    2,    1,    4
    15:    3,    2,    4,    1
    16:    3,    4,    1,    2
    17:    3,    4,    2,    1
    18:    4,    1,    2,    3
    19:    4,    1,    3,    2
    20:    4,    2,    1,    3
    21:    4,    2,    3,    1
    22:    4,    3,    1,    2
    23:    4,    3,    2,    1
```

```
Generating permutations using algorithm 7: Iterative order 4
     0:    1,    2,    3,    4
     1:    1,    2,    4,    3
     2:    1,    3,    2,    4
     3:    1,    3,    4,    2
     4:    1,    4,    2,    3
     5:    1,    4,    3,    2
     6:    2,    1,    3,    4
     7:    2,    1,    4,    3
     8:    2,    3,    1,    4
     9:    2,    3,    4,    1
    10:    2,    4,    1,    3
    11:    2,    4,    3,    1
    12:    3,    1,    2,    4
    13:    3,    1,    4,    2
    14:    3,    2,    1,    4
    15:    3,    2,    4,    1
    16:    3,    4,    1,    2
    17:    3,    4,    2,    1
    18:    4,    1,    2,    3
    19:    4,    1,    3,    2
    20:    4,    2,    1,    3
    21:    4,    2,    3,    1
    22:    4,    3,    1,    2
    23:    4,    3,    2,    1
Generating permutations using algorithm 8: Johnson-Trotter Recursive order 4
     0:    1,    2,    3,    4
     1:    1,    2,    4,    3
     2:    1,    4,    2,    3
     3:    4,    1,    2,    3
     4:    4,    1,    3,    2
     5:    1,    4,    3,    2
     6:    1,    3,    4,    2
     7:    1,    3,    2,    4
     8:    3,    1,    2,    4
     9:    3,    1,    4,    2
    10:    3,    4,    1,    2
    11:    4,    3,    1,    2
    12:    4,    3,    2,    1
    13:    3,    4,    2,    1
    14:    3,    2,    4,    1
    15:    3,    2,    1,    4
    16:    2,    3,    1,    4
    17:    2,    3,    4,    1
    18:    2,    4,    3,    1
    19:    4,    2,    3,    1
    20:    4,    2,    1,    3
    21:    2,    4,    1,    3
    22:    2,    1,    4,    3
    23:    2,    1,    3,    4
```

```
Generating permutations using algorithm 9: Johnson-Trotter Iterative order 4
     0:    1,    2,    3,    4
     1:    1,    2,    4,    3
     2:    1,    4,    2,    3
     3:    4,    1,    2,    3
     4:    4,    1,    3,    2
     5:    1,    4,    3,    2
     6:    1,    3,    4,    2
     7:    1,    3,    2,    4
     8:    3,    1,    2,    4
     9:    3,    1,    4,    2
    10:    3,    4,    1,    2
    11:    4,    3,    1,    2
    12:    4,    3,    2,    1
    13:    3,    4,    2,    1
    14:    3,    2,    4,    1
    15:    3,    2,    1,    4
    16:    2,    3,    1,    4
    17:    2,    3,    4,    1
    18:    2,    4,    3,    1
    19:    4,    2,    3,    1
    20:    4,    2,    1,    3
    21:    2,    4,    1,    3
    22:    2,    1,    4,    3
    23:    2,    1,    3,    4
```

11 Sorted permutations output for N = 4

```
data     807,  249,   73,  658
permute    1,    2,    3,    4
permuted 807,  249,   73,  658
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    1,    2,    4,    3
permuted 807,  249,  658,   73
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    1,    4,    2,    3
permuted 807,  658,  249,   73
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    4,    1,    2,    3
permuted 658,  807,  249,   73
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    4,    1,    3,    2
permuted 658,  807,   73,  249
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    1,    4,    3,    2
permuted 807,  658,   73,  249
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    1,    3,    4,    2
permuted 807,   73,  658,  249
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    1,    3,    2,    4
permuted 807,   73,  249,  658
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    3,    1,    2,    4
permuted  73,  807,  249,  658
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    3,    1,    4,    2
permuted  73,  807,  658,  249
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    3,    4,    1,    2
permuted  73,  658,  807,  249
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    4,    3,    1,    2
permuted 658,   73,  807,  249
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    4,    3,    2,    1
permuted 658,   73,  249,  807
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    3,    4,    2,    1
permuted  73,  658,  249,  807
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    3,    2,    4,    1
permuted  73,  249,  658,  807
sorted    73,  249,  658,  807
```

```
data     807,  249,   73,  658
permute    3,    2,    1,    4
permuted  73,  249,  807,  658
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    2,    3,    1,    4
permuted 249,   73,  807,  658
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    2,    3,    4,    1
permuted 249,   73,  658,  807
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    2,    4,    3,    1
permuted 249,  658,   73,  807
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    4,    2,    3,    1
permuted 658,  249,   73,  807
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    4,    2,    1,    3
permuted 658,  249,  807,   73
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    2,    4,    1,    3
permuted 249,  658,  807,   73
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    2,    1,    4,    3
permuted 249,  807,  658,   73
sorted    73,  249,  658,  807

data     807,  249,   73,  658
permute    2,    1,    3,    4
permuted 249,  807,   73,  658
sorted    73,  249,  658,  807
```