**October 2014**

CSC 342 – Exploration 3

James Francis/Sam Jentsch

Account Numbers: CSC342107/CSC342111

08

**Fall**

­­

Table of Contents

Problem Statement 3

Algorithm 3

Implementation 4

Java Implementation 4

C Implementation 4

Experiment 4

Machines 4

Timing Methods 5

Data Files and Sample Output 5

Charts and Timing Results 6

gprof & hprof 10

Conclusions 10

Future Work 10

Problems Encountered 10

Time Chart 10

Appendix A – C Johnson-Trotter 12

Appendix B – C Lexicographic Permute 14

Appendix C – Java Johnson-Trotter 16

Appendix D – Java Lexicographic Permute 18

Appendix E – References 20

# Problem Statement

The purpose of the assignment was to gain familiarity with algorithms for generating permutations using two well-known permutation algorithms, the Johnson-Trotter algorithm and the Lexicographic Permute algorithm.

Permutations are extremely useful in problems requiring the consideration of different choices. They can be used to model different possibilities in many problem scenarios and are extremely common in optimization problems, like the Traveling Salesman, Knapsack, and Assignment problems.

# Algorithm

The algorithms for both the Johnson-Trotter algorithm and the Lexicographic Permute algorithm were gathered from the text.

Johnson-Trotter:

JohnsonTrotter(n)

Initialize first permutation with { 1,…, n }

while the last permutation has a mobile element do

find its largest mobile element k

swap k with the adjacent element k’s arrow points to

reverse the direction of all the elements that are larger than k

add the new permutation to the list

In the Johnson-Trotter algorithm, each element in the permutation is marked with a directional arrow. This arrow indicates which elements in the permutation are mobile. An element in the permutation is mobile if its arrow is pointing to an element smaller than it is. Permutations are generated by swapping the largest mobile element, k, with the element its directional arrow points to. This continues until no more mobile elements exist. At this point all permutations have been generated.

Lexicographic Permute:

LexicographicPermute(n)

Initialize first permutation with 12…n

while last permutation has two consecutive elements in increasing order do

let i be its largest index such that a­­­­­­­­i < ai+1

find largest index j such that a­­­­­­­­i < aj

swap a­­­­­­­­i with aj

reverse order of elements from ai+1 to a­n

add new permutation to list

The Lexicographic Permute algorithm uses an index based representation of permutations a1a2a3…an. If the second to last element is less than the last element in the permutation (this is the case for half of all the permutations), these two elements can be swapped to generate the next permutation.

If the second to last element is greater than the last element, find the permutation’s longest decreasing suffix. The index of the element that is the beginning of this suffix will be denoted *i*. Increase the value at this index by exchanging it with the smallest element in the suffix. Finally, reverse the suffix so that it is in increasing order.

# Implementation

### Java Implementation

The Java implementation was found from Princeton University for both the Johnson-Trotter and Lexicographic Permute algorithms.

The implementations were straightforward and follow the algorithms presented in the class text.

### C Implementation

The C implementation for the Johnson-Trotter and Lexicographic Permute algorithms was found from University of Victoria.

Like the Java implementation, the implementations were simple to follow and implement the algorithms presented in the class text.

# Experiment

### Machines

The machines used for testing include:

* **15” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2GHz quad-core Intel core i7
* **13” Macbook Pro**
  + OS: OSX 10.10
  + MEMORY: 8 GB 1600 MHz DDR3
  + PROCESSOR: 2.4 GHz Intel Dual Core i5
* **SFA Unix Server**
  + OS: Linux 2.6.18-371.12.1.el5PAE i686
  + MEMORY: Undetermined (required sudo access)
  + PROCESSOR: eight-core Intel(R) Xeon(TM) MP CPU 3.66GHz

### Timing Methods

In the Java implementation, System.nanoTime() was utilized to find the system time before execution of the permutation algorithms. The system time was found in the same manner after the completion of the algorithms’ executions and subtracted from the start times. This method follows the timing method for Java described in the timing handout.

Timing was achieved in the C implementation of the program through the use of the <time.h> header. The system clock time was found before execution of the Johnson Trotter or Lexicographic Permute permutation call and the end time was subtracted from the start time to calculate the total execution time.

For timing, we removed the print methods from the implementations of both the Johnson-Trotter and Lexicographic Permute algorithms. This was to gather timings unbiased by the time required to output to the console.

### Data Files and Sample Output

Our sample data consisted of manually entering input values for both the Johnson-Trotter and Lexicographic permute methods on all machines tested. The results for input were recorded (number of permutations generated) along with the time it took for the permutations to be generated.

Johnson-Trotter:

Sample input: n = 3

Sample output:

123

132

312

321

231

213 🡪 Total *3! = 6 permutations*

Lexicographic Permute:

Sample input: n = 3

Sample Output:

123

132

213

231

312

321 🡪 Total *3! = 6 permutations*

### Charts and Timing Results

UNIX COMPARISONS

Charts

MAC COMPARISONS

### gprof & hprof

To do the limited number of function calls, we chose not to profile the algorithms for this exploration.

# Conclusions

The rate of growth for both algorithms for all permutations is exponential. This is what we expected since both algorithms generate permutations for a given input n. The number of permutations generated must be *n!* implying that the rate of growth for both algorithms is *O(n!)*.

In our tests, we concluded that the Johnson-Trotter algorithm is faster (at a minimum 1.32 times faster) at generating permutations than the Lexicographic Permute algorithm. We believe this is do to the additional overhead required in the Lexicographic Permute.

The Johnson-Trotter algorithm satisfies the minimal change requirement. This implies that each new permutation is generated by exchanging only two elements present in its immediate predecessor. The Lexicographic Permute on the other hand, requires the initial swap of the elements ai and aj, followed by additional swaps to reverse of the elements after ai before completing the next permutation.

Another observation we made is that the Java implementation of both algorithms was consistently faster than the C implementation for both systems. The Java implementation used was based off of the C implementation and in our previous experience the C implementation of the algorithms examined so far has been faster than the Java implementation. For lower values (less than n = 11), it appears that the C implementation is initially faster. This might be do to the reduced amount of overhead associated with C programs when compared to Java. For larger values of n, Java remained consistently faster. We speculate that this might be do to enhanced memory features present in Java like garbage collection.

### Future Work

One observation we made about the difference in permutations output between the Johnson-Trotter algorithm and Lexicographic Permute algorithms is that, since the Lexicographic algorithm generates permutations in increasing order, this would be the better choice for applications where a sorted list of permutations is beneficial (like if a binary search were performed on the permutations). Johnson-Trotter would require a sorting algorithm be applied to the permutation list generated to achieve the same effect.

### Problems Encountered

Timing for n=14, Lexicographic Permute in C on the server reported a negative number even though time was being represented by a long long int. We did not run into this issue for the C implementation of Lexicographic Permute on OS X or the Java implementation on UNIX and OS X.

### Time Chart

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Thursday | Friday | Saturday | Sunday | Monday | Tuesday | Wednesday |
| James | 2 hrs | 1.5 hrs | 0 hrs | 3 hrs | 2.5 hrs | 3.5 hrs | 9.5 hrs |
| Sam | 2 hrs | 1.5 hrs | 0 hrs | 3 hrs | 2.5 hrs | 3.5 hrs | 9 hrs |

# Appendix A – C Johnson-Trotter

/\*===================================================================\*/

/\* C program for distribution from the Combinatorial Object Server. \*/

/\* Generate permutations by transposing adjacent elements \*/

/\* via the Steinhaus-Johnson-Trotter algorithm. This is \*/

/\* the same version used in the book "Combinatorial Generation." \*/

/\* Both the permutation (in one-line notation) and the positions \*/

/\* being transposed (as a 2-cycle) are output. \*/

/\* The program can be modified, translated to other languages, etc., \*/

/\* so long as proper acknowledgement is given (author and source). \*/

/\* Programmer: Frank Ruskey, 1995. \*/

/\* The latest version of this program may be found at the site \*/

/\* <http://theory.cs.uvic.ca/inf/perm/PermInfo.html> \*/

/\*===================================================================\*/

#include <time.h>

#include <stdlib.h>

#include <stdio.h>

#define INT long long int

int NN, i, count=0 ;

int p[100], pi[100]; /\* The permutation and its inverse \*/

int dir[100]; /\* The directions of each element \*/

void PrintPerm() {

int i;

/\* uncomment if you want to print the index of each perm \*/

/\*

count = count + 1;

printf( "[%8d] ", count );

\*/

for (i=1; i <= NN; ++i) printf( "%d", p[i] );

} /\* PrintPerm \*/;

void PrintTrans( int x, int y ) {

printf( " (%d %d)", x, y ); printf( "\n" );

} /\* PrintTrans \*/;

void Move( int x, int d ) {

int z;

//PrintTrans( pi[x], pi[x]+d );

z = p[pi[x]+d];

p[pi[x]] = z;

p[pi[x]+d] = x;

pi[z] = pi[x];

pi[x] = pi[x]+d;

} /\* Move \*/;

void Perm ( int n ) {

int i;

if (n > NN) {

//PrintPerm();

} else {

Perm( n+1 );

for (i=1; i<=n-1; ++i) {

Move( n, dir[n] ); Perm( n+1 );

}

dir[n] = -dir[n];

}

} /\* of Perm \*/;

int main () {

printf( "Enter n: " ); scanf( "%d", &NN );

printf( "\n" );

for (i=1; i<=NN; ++i) {

dir[i] = -1; p[i] = i; pi[i] = i;

}

//start clock

clock\_t start = clock(), diff;

Perm ( 1 );

printf( "\n" );

//Clock end

diff = clock() - start;

INT msec = diff \* 1000 / CLOCKS\_PER\_SEC;

printf("Time taken %lli seconds %lli milliseconds\n\n", msec/1000, msec%1000);

}

# Appendix B – C Lexicographic Permute

/\*===================================================================\*/

/\* C program for distribution from the Combinatorial Object Server. \*/

/\* Generate permutations in lexicographic order. This is \*/

/\* the same version used in the book "Combinatorial Generation" by \*/

/\* Frank Ruskey. \*/

/\* The program can be modified, translated to other languages, etc., \*/

/\* so long as proper acknowledgement is given (author and source). \*/

/\* Programmer: Frank Ruskey 19??, Joe Sawada, 1997. \*/

/\* The latest version of this program may be found at the site \*/

/\* <http://theory.cs.uvic.ca/inf/perm/PermInfo.html> \*/

/\*===================================================================\*/

#include <time.h>

#include <stdlib.h>

#include <stdio.h>

#define INT long long int

int n;

int a[100]; /\* The permutation \*/

void PrintPerm() {

int i;

for (i=1; i <= n; i++) printf( "%d", a[i] );

printf("\n");

}

void swap(int i, int j) {

int temp;

temp = a[i];

a[i] = a[j];

a[j] = temp;

}

int Next() {

int k,j,r,s;

k = n-1;

while (a[k] > a[k+1]) k--;

if (k == 0) return(0);

else {

j = n;

while (a[k] > a[j]) j--;

swap(j,k);

r = n; s = k+1;

while (r>s) {

swap(r,s);

r--; s++;

}

}

//PrintPerm();

return(1);

}

int main () {

int i;

printf( "Enter n: " ); scanf( "%d", &n );

if (n<=0) exit(1);

printf( "\n" );

for (i=0; i<=n; ++i) {

a[i] = i;

}

//start clock

clock\_t start = clock(), diff;

//PrintPerm();

while (Next());

//printf( "\n" );

//Clock end

diff = clock() - start;

INT msec = diff \* 1000 / CLOCKS\_PER\_SEC;

printf("Time taken %lli seconds %lli milliseconds\n\n", msec/1000, msec%1000);

}

# Appendix C – Java Johnson-Trotter

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac Perm.java

\* Execution: java Permutations N k

\*

\* Generate permutations by transposing adjacent elements using the

\* Johnson-Trotter algorithm.

\*

\* This program is a Java version based on the program SJT.c

\* writen by Frank Ruskey.

\*

\* <http://theory.cs.uvic.ca/inf/perm/PermInfo.html>

\*

\* % java JohnsonTrotter 3

\* 012 (2 1)

\* 021 (1 0)

\* 201 (2 1)

\* 210 (0 1)

\* 120 (1 2)

\* 102 (0 1)

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class JohnsonTrotter {

public static void perm(int N) {

int[] p = new int[N]; // permutation

int[] pi = new int[N]; // inverse permutation

int[] dir = new int[N]; // direction = +1 or -1

for (int i = 0; i < N; i++) {

dir[i] = -1;

p[i] = i;

pi[i] = i;

}

perm(0, p, pi, dir);

//System.out.printf(" (0 1)\n");

}

public static void perm(int n, int[] p, int[] pi, int[] dir) {

// base case - print out permutation

if (n >= p.length) {

//for (int i = 0; i < p.length; i++)

// System.out.print(p[i]);

return;

}

perm(n+1, p, pi, dir);

for (int i = 0; i <= n-1; i++) {

// swap

//System.out.printf(" (%d %d)\n", pi[n], pi[n] + dir[n]);

int z = p[pi[n] + dir[n]];

p[pi[n]] = z;

p[pi[n] + dir[n]] = n;

pi[z] = pi[n];

pi[n] = pi[n] + dir[n];

perm(n+1, p, pi, dir);

}

dir[n] = -dir[n];

}

public static void main(String[] args) {

int N = Integer.parseInt(args[0]);

//start time

long startTime = System.nanoTime();

long stopTime = System.nanoTime();

long overTime = stopTime - startTime;

startTime = System.nanoTime();

perm(N);

stopTime = System.nanoTime();

//Stop time for the algorithm

long time = stopTime - startTime - overTime;

System.out.println("Total time for "+N+" is " +(time/1000000));

}

}

# Appendix D – Java Lexicographic Permute

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Compilation: javac PermutationsLex.java

\* Execution: java PermutationsLex N

\*

\* Generate all N! permutations of N elements in lexicographic order.

\*

\* This program is a Java version based on the program Permlex.c

\* writen by Frank Ruskey and Joe Sawada.

\*

\* <http://theory.cs.uvic.ca/inf/perm/PermInfo.html>

\*

\* % java PermutationsLex 3

\* 012

\* 021

\* 102

\* 120

\* 201

\* 210

\*

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

public class PermutationsLex {

public static void show(int[] a) {

for (int i = 0; i < a.length; i++)

StdOut.printf("%d", a[i]);

StdOut.printf("\n");

}

public static void swap(int[] a, int i, int j) {

int temp = a[i];

a[i] = a[j];

a[j] = temp;

}

public static boolean hasNext(int[] a) {

int N = a.length;

// find rightmost element a[k] that is smaller than element to its right

int k;

for (k = N-2; k >= 0; k--)

if (a[k] < a[k+1]) break;

if (k == -1) return false;

// find rightmost element a[j] that is larger than a[k]

int j = N-1;

while (a[k] > a[j])

j--;

swap(a, j, k);

for (int r = N-1, s = k+1; r > s; r--, s++)

swap(a, r, s);

return true;

}

public static void perm(int N) {

// initialize permutation

int[] a = new int[N];

for (int i = 0; i < N; i++)

a[i] = i;

// print permutations

//show(a);

while (hasNext(a)){

//show(a);

}

}

public static void main(String[] args) {

int N = Integer.parseInt(args[0]);

//start time

long startTime = System.nanoTime();

long stopTime = System.nanoTime();

long overTime = stopTime - startTime;

startTime = System.nanoTime();

perm(N);

stopTime = System.nanoTime();

//Stop time for the algorithm

long time = stopTime - startTime - overTime;

System.out.println("Total time for "+N+" is " +(time/1000000));

}

}

# Appendix E – References

<http://introcs.cs.princeton.edu/java/23recursion/PermutationsLex.java.html>

<http://introcs.cs.princeton.edu/java/23recursion/JohnsonTrotter.java.html>

<http://theory.cs.uvic.ca/inf/perm/PermInfo.html>

Introduction to the Design and Analysis of Algorithms by Anany Levitin

Assignment sheet – Dr. Robert Strader