PERMUTATION GENERATION METHODS

Robert Sedgewick
Princeton University
**Motivation**

**PROBLEM**  Generate all N! permutations of N elements

Q: Why?
- Basic research on a fundamental problem
- Compute exact answers for insights into combinatorial problems
- Structural basis for backtracking algorithms

Numerous published algorithms, dating back to 1650s

**CAVEATS**
- N is between 10 and 20
- can be the basis for extremely dumb algorithms
- processing a perm often costs much more than generating it
N is between 10 and 20

<table>
<thead>
<tr>
<th>N</th>
<th>number of perms</th>
<th>million/sec</th>
<th>billion/sec</th>
<th>trillion/sec</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3628800</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>39916800</td>
<td>seconds</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>479001600</td>
<td>minutes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>6227020800</td>
<td>hours</td>
<td>seconds</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>87178291200</td>
<td>day</td>
<td>minute</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>1307674368000</td>
<td>weeks</td>
<td>minutes</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>20922789888000</td>
<td>months</td>
<td>hours</td>
<td>seconds</td>
</tr>
<tr>
<td>17</td>
<td>355687428096000</td>
<td>years</td>
<td>days</td>
<td>minutes</td>
</tr>
<tr>
<td>18</td>
<td>6402373705728000</td>
<td></td>
<td>months</td>
<td>hours</td>
</tr>
<tr>
<td>19</td>
<td>121645100408832000</td>
<td></td>
<td>years</td>
<td>days</td>
</tr>
<tr>
<td>20</td>
<td>2432902008176640000</td>
<td></td>
<td></td>
<td>month</td>
</tr>
</tbody>
</table>
Digression: analysis of graph algorithms

Typical graph-processing scenario:

- input graph as a sequence of edges (vertex pairs)
- build adjacency-lists representation
- run graph-processing algorithm

Q: Does the order of the edges in the input matter?
A: Of course!

Q: How?
A: It depends on the graph

Q: How?

There are $2^{V^2}$ graphs, so full employment for algorithm analysts
Ex: compute a spanning forest (DFS, stop when $V$ vertices hit)

best case cost: $V$ (right edge appears first on all lists)

Complete digraph on $V$ vertices

worst case: $V^2$

average: $V \ln V$ (Kapidakis, 1990)

Same graph with single outlier

worst case: $O(V^2)$

average: $O(V^2)$

Can we estimate the average for a given graph?

Is there a simple way to reorder the edges to speed things up?

What impact does edge order have on other graph algorithms?
Digression: analysis of graph algorithms

Insight needed, so generate perms to study graphs

No shortage of interesting graphs with fewer than 10 edges

Algorithm to compute average
  ◦ generate perms, run graph algorithm

Goal of analysis
  ◦ faster algorithm to compute average
Method 1: backtracking

Compute all perms of a global array by exchanging each element to the end, then recursively permuting the others

```c
exch (int i, int j)
{ int t = p[i]; p[i] = p[j]; p[j] = t; }
generate(int N)
{ int c;
  if (N == 1) doit();
  for (c = 1; c <= N; c++)
  { exch(c, N); generate(N-1); exch(c, N); }
}
```

Invoke by calling

```
generate(N);
```

**Problem:** Too many (2N!) exchanges (!)
Method 2: “Plain changes”

Sweep first element back and forth to insert it into every position in each perm of the other elements

Generates all perms with $N!$ exchanges of adjacent elements

Dates back to 1650s (bell ringing patterns in English churches)

Exercise: recursive implementation with constant time per exch
Eliminate first exch in backtracking

```c
exch (int i, int j)
{ int t = p[i]; p[i] = p[j]; p[j] = t; }
generate(int N)
{ int c;
  if (N == 1) doit();
  for (c = 1; c <= N; c++)
    { generate(N-1); exch(?, N); } }
```

**Detail(?)**: Where is new item for p[N] each time?
**Index table computation**

**Q:** how do we find a new element for the end?

**A:** compute an index table from the (known) perm for N-1

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>A</td>
<td>C</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

so all perms of 3 takes into

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
</tbody>
</table>

so all perms of 4 takes into

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
<td>A</td>
</tr>
</tbody>
</table>

and so forth

**Exercise:** Write a program to compute this table
Method 3: general recursive single-exch

Use precomputed index table
Generates perms with N! exchanges
Simple recursive algorithm

generate(int N)
{
    int c;
    if (N == 1) doit();
    for (c = 1; c <= N; c++)
    {
        generate(N-1); exch(B[N][c], N);
    }
}

No need to insist on particular sequence for last element

specifies (N − 1)! (N − 2)!...3!2! different algorithms

Table size is N(N-1)/2 but N is less than 20

Do we need the table?
Method 4: Heap’s* algorithm

Index table is not needed

Q: where can we find the next element to put at the end?

A: at 1 if N is odd; i if N is even

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>A</td>
<td>C</td>
<td>C</td>
<td>B</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>B</td>
<td>B</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>D</td>
<td>D</td>
<td>C</td>
<td>C</td>
<td>B</td>
</tr>
</tbody>
</table>

Exercise: Prove that it works!

*Note: no relationship between Heap and heap data structure
Implementation of Heap's method (recursive)

Simple recursive function

```c
generate(int N)
{
    int c;
    if (N == 1) { doit(); return; }
    for (c = 1; c <= N; c++)
    {
        generate(N-1);
        exch(N % 2 ? 1 : c, N)
    }
}
```

N! exchanges
Starting point for code optimization techniques
Simple recursive function easily adapts to backtracking

```c
generate(int N) {
    int c;
    if (test(N)) return;
    for (c = 1; c <= N; c++) {
        generate(N-1);
        exch(N % 2 ? 1 : c, N)
    }
}
```

N! exchanges saved when test succeeds
Factorial counting

Count using a mixed-radix number system

```c
for (n = 1; n <= N; n++)
    c[n] = 1;
for (n = 1; n <= N; )
    if (c[n] < n) { c[n]++; n = 1; }
    else c[n++] = 1;
```

Values of digit $i$ range from 1 to $i$

(Can derive code by systematic recursion removal)

1-1 correspondence with permutations

- commonly used to generate random perms
  ```c
  for (i = 1; i <=N i++) exch(i, random(i));
  ```

Use as control structure to generate perms
Implementation of Heap's method (nonrecursive)

generate(int N)
{
    int n, t, M;
    for (n = 1; n <= N; n++)
    {
        p[n] = n; c[n] = 1;
    }
    doit();
    for (n = 1; n <= N; )
    {
        if (c[n] < n)
        {
            exch(N % 2 ? 1 : c, N)
            c[n]++; n = 1;
            doit();
        } else c[n++] = 1;
    }
}

"Plain changes" and most other algs also fit this schema
Analysis of Heap's method

Most statements are executed $N!$ times (by design) except

$B(N)$: the number of tests for $N$ equal to 1 (loop iterations)

$A(N)$: the extra cost for $N$ odd

Recurrence for $B$

$B(N) = NB(N-1) + 1$ for $N > 1$ with $B(1) = 1$

Solve by dividing by $N!$ and telescoping

$$\frac{B(N)}{N!} = \frac{B(N-1)}{(N-1)!} + \frac{1}{N!} = 1 + \frac{1}{2!} + \frac{1}{3!} + \ldots + \frac{1}{N!}$$

Therefore $B(N) = \left\lfloor N!(e - 1) \right\rfloor$ and similarly $A(N) = \left\lfloor N!/e \right\rfloor$

Typical running time: $19N! + A(N) + 10B(N) \approx 36.55N!$

worthwhile to lower constant huge quantity
Improved version of Heap's method (recursive)

```c
generate(int N)
{
    int c;
    if (N == 3)
    {
        doit();
        p1 = p[1]; p2 = p[2]; p3 = p[3];
    }
    for (c = 1; c <= N; c++)
    {
        generate(N-1);
        exch(N % 2 ? 1 : c, N)
    }
}
N! exchanges
Starting point for code optimization techniques
Improved version of Heap's method (recursive)
Bottom line

Quick empirical study on this machine (N = 12)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Heap (recursive)! !</td>
<td>415.2</td>
</tr>
<tr>
<td>cc -O4!!</td>
<td>54.1</td>
</tr>
<tr>
<td>Java!!</td>
<td>442.8</td>
</tr>
<tr>
<td>Heap (nonrecursive)! !</td>
<td>84.0</td>
</tr>
<tr>
<td>inline N = 2!!</td>
<td>92.4</td>
</tr>
<tr>
<td>inline N = 3!!</td>
<td>51.7</td>
</tr>
<tr>
<td>cc -O4!!</td>
<td>3.2</td>
</tr>
</tbody>
</table>

about (1/6) billion perms/second

Lower Bound: about 2N! register transfers
References

Heap, “Permutations by interchanges,”
Computer Journal, 1963

Knuth, The Art of Computer Programming, vol. 4 sec. 7.2.1.1
//www-cs-faculty.stanford.edu/~knuth/taocp.html

Ord-Smith, “Generation of permutation sequences,”
Computer Journal, 1970-71

Sedgewick, Permutation Generation Methods,
Computing Surveys, 1977

Trotter, “Perm (Algorithm 115),”
CACM, 1962

Wells, Elements of combinatorial computing, 1961
[see surveys for many more]
Digression: analysis of graph algorithms

Initial results (Dagstuhl, 2002)