Now is the time
For all good men
To come to the aid
Of their party

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

Can be the basis for extremely dumb algorithms

Processing a perm often costs much more than generating it

**Caveats**

$N$ is between 10 and 20

Numerous published algorithms, dating back to 1650s
<table>
<thead>
<tr>
<th>Month</th>
<th>N is between 10 and 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Days</td>
<td></td>
</tr>
<tr>
<td>Years</td>
<td></td>
</tr>
<tr>
<td>Hours</td>
<td></td>
</tr>
<tr>
<td>Months</td>
<td></td>
</tr>
<tr>
<td>Minutes</td>
<td></td>
</tr>
<tr>
<td>Days</td>
<td></td>
</tr>
<tr>
<td>Years</td>
<td></td>
</tr>
<tr>
<td>Months</td>
<td></td>
</tr>
<tr>
<td>Hours</td>
<td></td>
</tr>
<tr>
<td>Minutes</td>
<td></td>
</tr>
<tr>
<td>Weeks</td>
<td></td>
</tr>
<tr>
<td>Minutes</td>
<td></td>
</tr>
<tr>
<td>Hours</td>
<td></td>
</tr>
<tr>
<td>Seconds</td>
<td></td>
</tr>
<tr>
<td>Number of perms</td>
<td></td>
</tr>
</tbody>
</table>

N is between 10 and 20
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: It depends on the graph.
Q: How?
A: Of course.

There are $2^{|V|^2}$ graphs, so full employment for algorithm analysts.
What impact does edge order have on other graph algorithms?

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

Can we estimate the average for a given graph?

- **Complete digraph on** $V$ **vertices**
  - **Average**: $O(V^2)$
  - **Worst case**: $O(V^2)$

- **Same graph with single outlier**
  - **Average**: $V \ln V$ (Kapidakis, 1990)
  - **Worst case**: $V^2$

- **Complete digraph on** $V$ **vertices**
  - **Best case cost**: $V$ (right edge appears first on all lists)

**Ex:** compute a spanning forest (DFS, stop when $V$ vertices hit)

Digression (continued)
Insight needed, so generate perms to study graphs.

No shortage of interesting graphs with fewer than 10 edges.
Problem: Too many (2N!) exchanges

Invoke by calling:

```
{ exch(c, N); generate(N-1); exch(c, N); }
```

```
genenerate(N);
```

Method 1: Backtracking

Compute all perms of a global array by exchanging each element to the end, then recursively permuting the others, then reversing the array.
Method 2: "Plain changes"

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

A  B  B  B  C  C  C  A  A  C  C  C  D  D  D  A  A  D  D  D  B  B  B  A
B  A  C  C  B  B  A  C  C  A  D  D  C  C  A  D  D  A  B  B  D  D  A  B
C  C  A  D  D  A  B  B  D  D  A  B  B  A  C  C  B  B  A  C  C  A  D  D
D  D  D  A  A  D  D  D  B  B  B  A  A  B  B  B  C  C  C  A  A  C  C  C

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 exchange.
Eliminate first exch in backtracking

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

General single-exch recursive scheme
Exercise: Write a program to compute this table and so forth

so all perms of 4 takes into

so all perms of 3 takes into

so all perms of 2 takes into

so all perms of 1 takes into

A: Compute an index table from the (known) perm for N-1
Q: How do we find a new element for the end?

Index Table Computation

Q: how do we find a new element for the end?
Method 3: general recursive single-exchange

Use precomputed index table

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

Specifics (N-I)!(N-2)!. . .3!2! different algorithms

No need to insist on particular sequence for last element

Simple recursive algorithm

Generates perms with N! exchanges

Do we need the table?
Method 4: Heap's* Algorithm

Index table is not needed.

Exercise: Prove that it works!

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

*Note: no relationship between Heap and heap data structure
Simple recursive function

Implementation of Heap's method (recursive)

Starting point for code optimization techniques

N! exchanges

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

```c
for (c = 1; c <= N; c++)
{
    if (N == 1) { doit(); return; }
    if (N != 1)
    {
        int c;
        generate(N-1)
    }
    exch(N % 2 ? 1 : c, N)
generate(N-1)
```
Simple recursive function easily adapts to backtracking

Implemententation of Heap's method (recursive)

\[ \text{generate} \text{if test succeeds} \]

\[
\begin{align*}
\text{generate}(\text{int } N) \quad &\{ \\
\text{int } c; \\
\text{if } (\text{test}(N)) \quad &\text{return}; \\
\text{for } (c = 1; c <= N; c++) \\
\text{exchange}(N \% 2 ? 1 : c, N); \\
\text{generate}(N-1); \\
\} \\
\end{align*}
\]
Factorial counting

Count using a mixed-radix number system

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.

1-1 correspondence with permutations

(commonly used to generate random perms)
(can derive code by systematic recursion removal)

Use as control structure to generate perms

\(\text{for (i = 1; i <= N; i++) exch(i, random(i));}\)

(use as control structure to generate perms)

1-1 correspondence with permutations

(count using a mixed-radix number system)

Facto...
Implementation of Heap's method (nonrecursive)
Most statements are executed $N!$ times (by design) except $B(N)$:

- the number of tests for $N$ equal to 1 (loop iterations)
- the extra cost for $N$ odd

Recurrence for $B(N)$:

\[
B(N) = NB(N-1) + 1, \quad \text{for } N > 1
\]

Solve by dividing by $N!$ and telescoping:

\[
\frac{i}{1} + \frac{i}{2} + \cdots + \frac{i}{i} = \frac{i}{1} + \frac{i(i-1)}{(N-1)N} = \frac{i}{N!} = \left(\frac{N(e-1)}{N!}\right)
\]

Therefore

\[
B(N) = N!(e-1)
\]

and similarly

\[
A(N) = N!/e
\]

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

Analysis of Heap's method

- worthwhile to lower constant
- huge quantity
- worthwhile to lower constant

Most statements are executed $N!$ times (by design) except $B(N)$. The extra cost for $N$ odd.
```c
{ 
    \text{generate}(\text{int} \, N) 
    \{ 
        \text{int} \, c; 
        \text{if} \ (N == 3) \{ 
            \text{doit();} 
            p1 = p[1]; \ p2 = p[2]; \ p3 = p[3]; 
            p[2] = p1; \ p[1] = p2; \ \text{doit();} 
            p[1] = p3; \ p[3] = p2; \ \text{doit();} 
            p[1] = p2; \ p[3] = p1; \ \text{doit();} 
            p[1] = p3; \ p[2] = p2; \ \text{doit();} 
            \text{return;}
        \}
        \text{for} \ (c = 1; \ c <= N; \ c++) 
        \{ 
            \text{generate}(N-1); 
            \text{exch}(N \% 2 ? 1 : c, N); 
        \}
    \}
    \text{for} \ (c = 1; \ c <= N; \ c++) 
    \{ 
        \text{doit();} 
        \text{return;}
    \}
}\text{Improved version of Heap's method (recursive)}
```
Quick empirical study on this machine (N = 12)

<table>
<thead>
<tr>
<th></th>
<th>cc -O4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inline</td>
<td>N = 3</td>
</tr>
<tr>
<td></td>
<td>3.2 secs</td>
</tr>
<tr>
<td>Inline</td>
<td>N = 2</td>
</tr>
<tr>
<td></td>
<td>5.7 secs</td>
</tr>
<tr>
<td>442.8 secs</td>
<td>Heap (nonrecursive)</td>
</tr>
<tr>
<td>442.8 secs</td>
<td>Heap</td>
</tr>
<tr>
<td>492.4 secs</td>
<td>cc -O4i</td>
</tr>
<tr>
<td>54.1 secs</td>
<td>cc -O4i</td>
</tr>
<tr>
<td>415.2 secs</td>
<td>Heap (recursive)</td>
</tr>
</tbody>
</table>
References

Wells, Elements of Combinatorial Computing, 1961
Heap, "Permutations by interchanges," CACM, 1962
Trotter, "Perm (Algorithm I15)," Computing Surveys, 1977
Knuth, The Art of Computer Programming, vol. 4, sec. 7.2.1.1

[see surveys for many more]
Digression: analysis of graph algorithms

Initial results (Dagstuhl, 2002)