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

N	number of perms	million/sec	billion/sec	trillion/sec
10	3628800			
11	39916800	seconds		
12	479001600	minutes		
13	6227020800	hours	seconds	
14	87178291200	day	minute	
15	1307674368000	weeks	minutes	
16	20922789888000	months	hours	seconds
17	355687428096000	years	days	minutes
18	6402373705728000		months	hours
19	121645100408832000		years	days
20	2432902008176640000			month

insignificant

impossible

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?

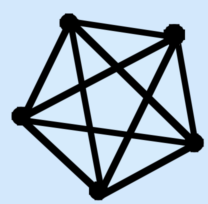
There are 2^{V^2} graphs, so full employment for algorithm analysts

Digression (continued)

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

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

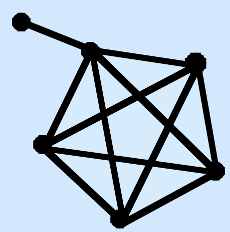
Complete digraph on V vertices



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?

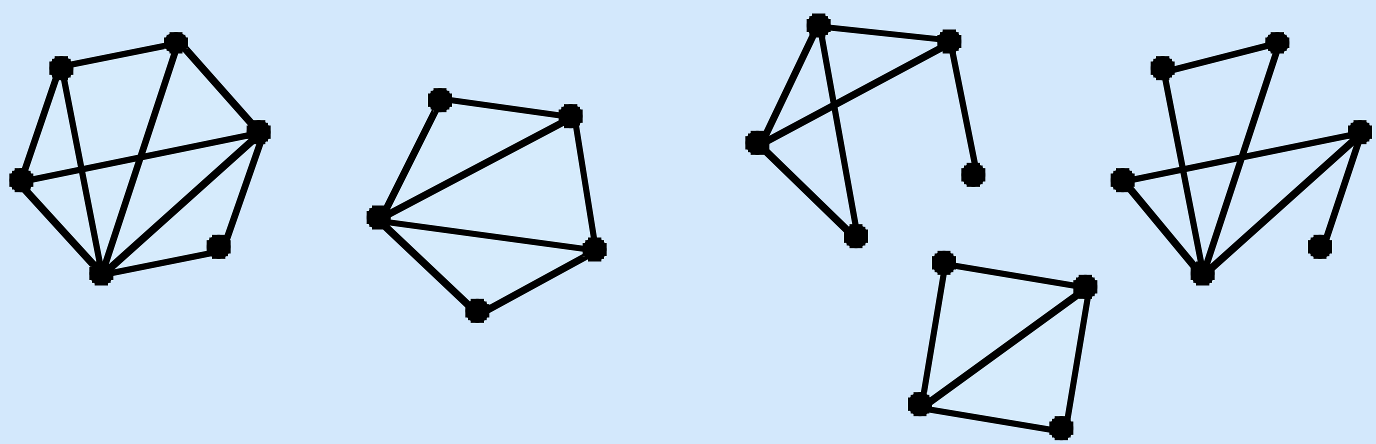
Is there a simple way to reorder the edges to speed things up?

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

```
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

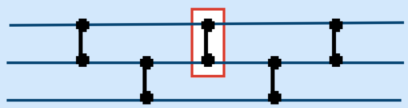
```
generate(N);
```

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

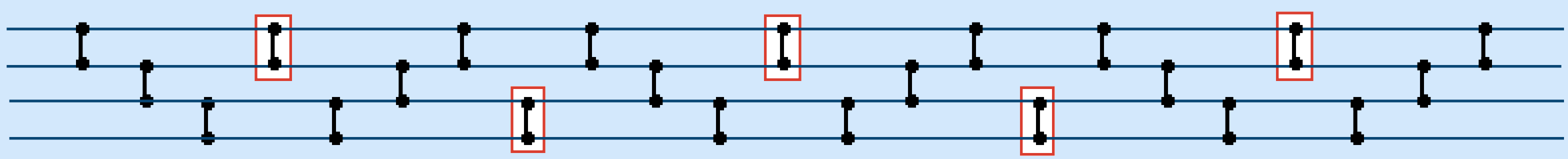
Problem: Too many (2N!) exchanges (!)

Method 2: "Plain changes"

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



A	B	B	C	C	A
B	A	C	B	A	C
C	C	A	A	B	B



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 exch

General single-exch recursive scheme

Eliminate first exch in backtracking

```
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

A	B	C	A	B	C
B	A	A	C	C	B
C	C	B	B	A	A
	1		1		

so all perms of 3 takes

A
B
C

into

C
B
A

A	C	D	A	A	D	D	B
B	B	B	B	C	C	C	C
C	A	A	D	D	A	B	D
D	D	C	C	B	B	A	A
	1		2		3		

so all perms of 4 takes

A
B
C
D

into

B
C
D
A

A	B	B	C	D	E	E	A	B	C
B	C	C	E	E	A	A	C	C	D
C	D	E	A	A	B	C	D	D	E
D	A	A	B	B	D	D	E	E	B
E	E	D	D	C	C	B	B	A	A
	3		1		3		1		

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); }
  }
```

1	1								
1	2	3							
3	1	3	1						
3	4	3	2	3					
5	3	1	5	3	1				
5	2	7	2	1	2	3			
7	1	5	5	3	3	7	1		
7	8	1	6	5	4	9	2	3	
9	7	5	3	1	9	7	5	3	1

No need to insist on particular sequence for last element

- specifies $(N-1)!\,(N-2)!\ldots3!\,2!$ different algorithms

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

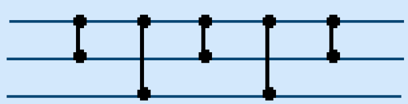
Do we need the table?

Method 4: Heap's* algorithm

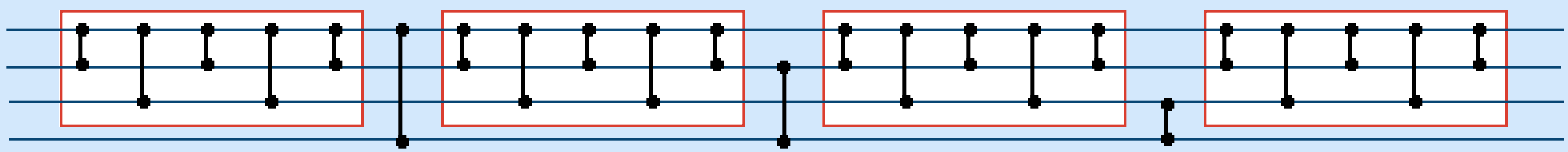
Index table is not needed

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



A	B	C	A	B	C
B	A	A	C	C	B
C	C	B	B	A	A



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

Exercise: Prove that it works!

*Note: no relationship between Heap and heap data structure

Implementation of Heap's method (recursive)

Simple recursive function

```
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

Implementation of Heap's method (recursive)

Simple recursive function easily adapts to backtracking

```
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

```
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

```
for (i = 1; i <=N i++) exch(i, random(i));
```

Use as control structure to generate perms

ABCD
BACD
BACD
BDCA

```
1111
1211
1121
1221
1131
1231
1112
1212
1122
1222
1132
1232
1113
1213
1123
1223
1133
1233
1114
1214
1124
1224
1134
1234
```

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

$$B(N) = NB(N-1) + 1 \quad \text{for } N > 1 \text{ with } B(1) = 1$$

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 $\boxed{B(N) = \lfloor N!(e-1) \rfloor}$ and similarly $\boxed{A(N) = \lfloor N!/e \rfloor}$

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

worthwhile to lower constant   huge quantity

Improved version of Heap's method (recursive)

```
generate(int N)
  { int c;
    if (N == 3)
      { doit();
        p1 = p[1]; p2 = p[2]; p3 = p[3];
        p[2] = p1; p[1] = p2; doit();
        p[1] = p3; p[3] = p2; doit();
        p[1] = p1; p[2] = p3; doit();
        p[1] = p2; p[3] = p1; doit();
        p[1] = p3; p[2] = p2; doit(); return;
      }
    for (c = 1; c <= N; c++)
      {
        generate(N-1);
        exch(N % 2 ? 1 : c, N)
      }
  }
```

Bottom line

Quick empirical study on this machine (N = 12)

Heap (recursive)	415.2 secs
cc -O4	54.1 secs
Java	442.8 secs
Heap (nonrecursive)	84.0 secs
inline N = 2	92.4 secs
inline N = 3	51.7 secs
cc -O4	3.2 secs

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)