Combinatorial Search

- permutations
- backtracking
- counting
- subsets
- paths in a graph

Overview

Exhaustive search. Iterate through all elements of a search space.

Backtracking. Systematic method for examining feasible solutions to a problem, by systematically eliminating infeasible solutions.

Applicability. Huge range of problems (include NP-hard ones).

Caveat. Search space is typically exponential in size $\Rightarrow$ effectiveness may be limited to relatively small instances.

Caveat to the caveat. Backtracking may prune search space to reasonable size, even for relatively large instances

Warmup: enumerate N-bit strings

Problem: process all 2^N N-bit strings (stay tuned for applications).

Equivalent to counting in binary from 0 to 2^N - 1.

- maintain `a[i]` where `a[i]` represents bit `i`
- initialize all bits to `0`
- simple recursive method does the job (call `enumerate(0)`)

```
private void enumerate(int k)
{
  if (k == N)
  {  process(); return;  }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[k] = 0;
}
```

clean up

starts with all 0s

```
0 0 0 0
0 0 0 1
0 0 1 0
0 0 1 1
0 1 0 0
0 1 0 1
0 1 1 0
0 1 1 1
1 0 0 0
1 0 0 1
1 0 1 0
1 0 1 1
1 1 0 0
1 1 0 1
1 1 1 0
1 1 1 1
```

```
1 1 1 0
1 1 0 0
1 0 0 0
0 0 0 0
```

example showing cleanups that zero out digits

ends with all 0s

Invariant (prove by induction);

Enumerates all (N-k)-bit strings and cleans up after itself.

Warmup: enumerate N-bit strings (full implementation)

Equivalent to counting in binary from 0 to $2^N - 1$.

```
public class Counter
{
   private int N;   // number of bits
   private int[] a; // bits (0 or 1)

   public Counter(int N)
   {
      this.N = N;
      a = new int[N];
      for (int i = 0; i < N; i++)
          a[i] = 0;
      enumerate(0);
   }

   private void enumerate(int k)
   {
     if (k == N)
     {  process(); return;  }
     enumerate(k+1);
     a[k] = 1;
     enumerate(k+1);
     a[k] = 0;
   }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Counter c = new Counter(N);
   }
}
```

optional (in this case)

all the programs in this lecture are variations on this theme

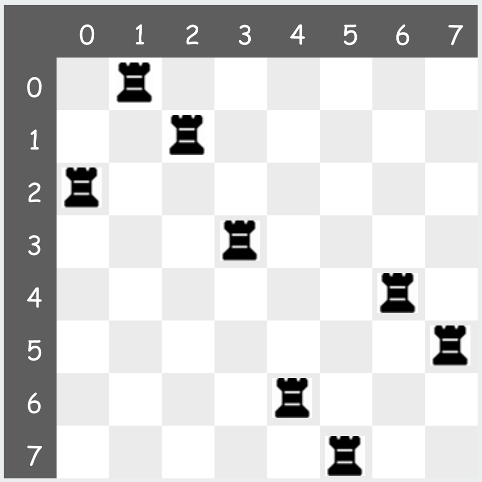
```
private void process()
{
   for (int i = 0; i < N; i++)
      StdOut.print(a[i]);
   StdOut.println();
}
```

```
% java Counter 4
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111
```

- permutations
- backtracking
- counting
- subsets
- paths in a graph

N-rooks Problem

How many ways are there to place
N rooks on an N-by-N board so that no rook can attack any other?



original problem: N = 8

```
int[] a = { 1, 2, 0, 3, 6, 7, 4, 5 };
```

No two in the same row, so represent solution with an array
`a[i]` = column of rook in row `i`.

No two in the same column, so array entries are all different
`a[]` is a permutation (rearrangement of 0, 1, ... N-1)

Answer: There are N! non mutually-attacking placements.

Challenge: Enumerate them all.

Enumerating permutations

Recursive algorithm to enumerate all N! permutations of size N:

- Start with `0 1 2 ... N-1`.
- For each value of `i`
  - swap `i` into position `0`
  - enumerate all (N-1)! arrangements of `a[1..N-1]`
  - clean up (swap `i` and `0` back into position)

```
0 1
1 0
```

```
0 1 2
0 2 1
1 0 2
1 2 0
2 1 0
2 0 1
```

0 followed by perms of 1 2 3	1 followed by perms of 0 2 3	2 followed by perms of 1 0 3	3 followed by perms of 1 2 0
0 1 2 3	1 0 2 3	2 1 0 3	3 1 2 0
0 1 3 2	1 0 3 2	2 1 3 0	3 1 0 2
0 2 1 3	1 2 0 3	2 0 1 3	3 2 1 0
0 2 3 1	1 2 3 0	2 0 3 1	3 2 0 1
0 3 2 1	1 3 2 0	2 3 0 1	3 0 2 1
0 3 1 2	1 3 0 2	2 3 1 0	3 0 1 2
	1 3 2 0		
	1 0 2 3		
	0 1 2 3		

example showing cleanup swaps that bring perm back to original

N-rooks problem (enumerating all permutations): *scaffolding*

```
public class Rooks
{
   private int N;
   private int[] a;

   public Rooks(int N)
   {
      this.N = N;
      a = new int[N];
      for (int i = 0; i < N; i++)
         a[i] = i;
      enumerate(0);
   }

   private void enumerate(int k)
   { /* See next slide. */  }

   private void exch(int i, int j)
   {  int t = a[i]; a[i] = a[j]; a[j] = t;  }

   private void process()
   {
      for (int i = 0; i < N; i++)
          StdOut.print(a[i] + " ");
      StdOut.println();
   }

   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      Rooks t = new Rooks(N);
      t.enumerate(0);
   }
}
```

initialize **a[0..N-1]** to **0..N-1**

```
% java Rooks 3
0 1 2
0 2 1
1 0 2
1 2 0
2 1 0
2 0 1
```

N-rooks problem (enumerating all permutations): recursive enumeration

Recursive algorithm to enumerate all N! permutations of size N:

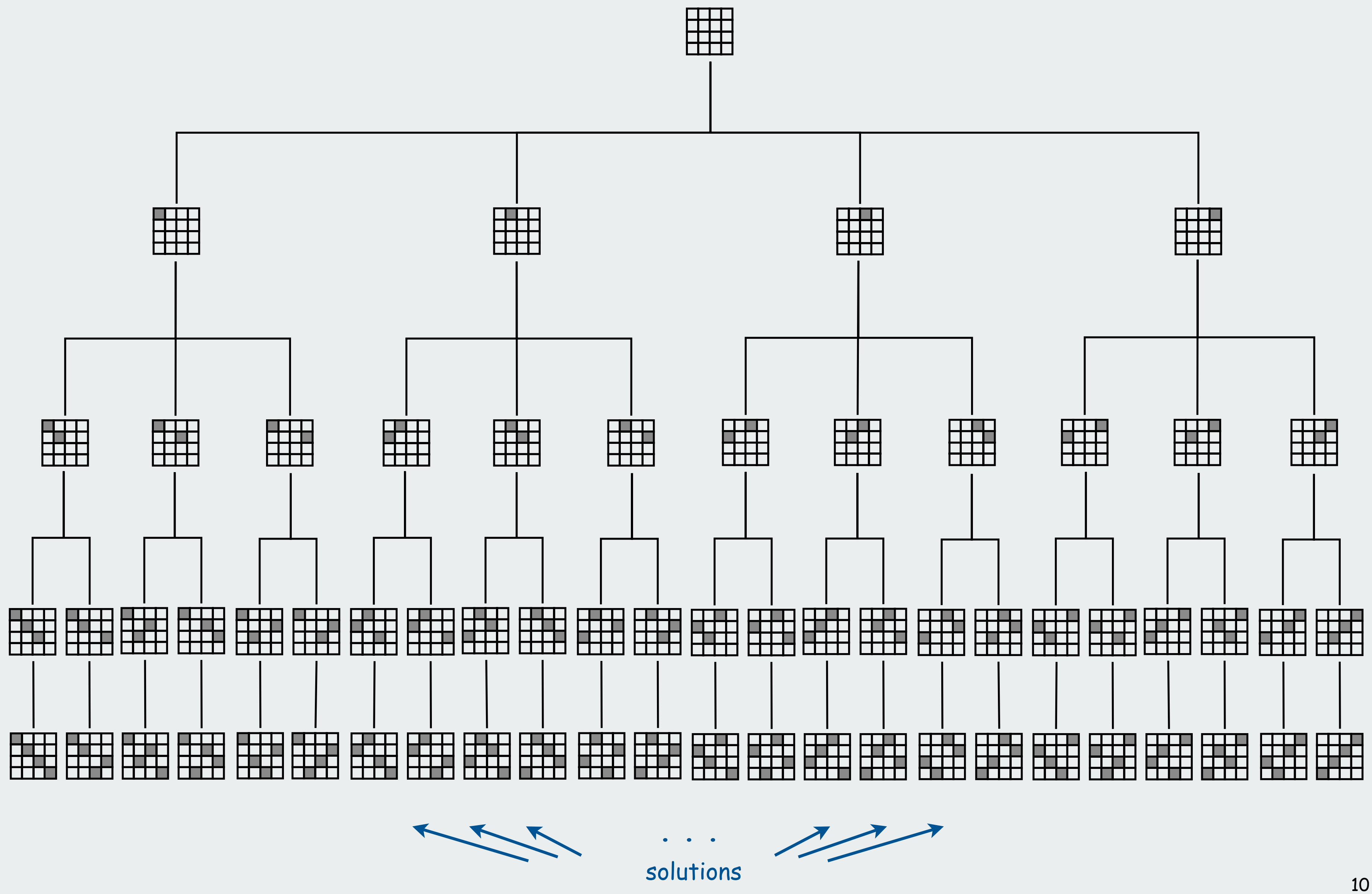
- Start with `0 1 2 ... N-1.`
- For each value of `i`
  - swap `i` into position `0`
  - enumerate all (N-1)! arrangements of `a[1..N-1]`
  - clean up (swap `i` and `0` back into position)

```
private void enumerate(int k)
{
   if (k == N)
   {
      process();
      return;
   }
   for (int i = k; i < N; i++)
   {
      exch(a, k, i);
      enumerate(k+1);
      exch(a, k, i);
   }
}
```

clean up

```
% java Rooks 4
0 1 2 3
0 1 3 2
0 2 1 3
0 2 3 1
0 3 2 1
0 3 1 2
1 0 2 3
1 0 3 2
1 2 0 3
1 2 3 0
1 3 2 0
1 3 0 2
2 1 0 3
2 1 3 0
2 0 1 3
2 0 3 1
2 3 0 1
2 3 1 0
3 1 2 0
3 1 0 2
3 2 1 0
3 2 0 1
3 0 2 1
3 0 1 2
```

4-Rooks search tree



N-rooks problem: back-of-envelope running time estimate

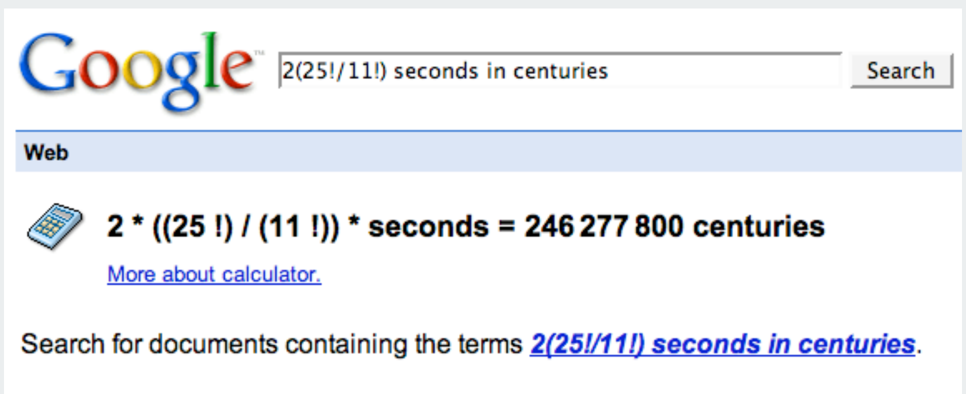
[Studying slow way to compute N! but good warmup for calculations.]

```
% java Rooks 10
3628800 solutions
```
instant

```
% java Rooks 11
39916800 solutions
```
about 2 seconds

```
% java Rooks 12
479001600 solutions
```
about 24 seconds (checks with N! hypothesis)

Hypothesis: Running time is about 2(N! / 11!) seconds.

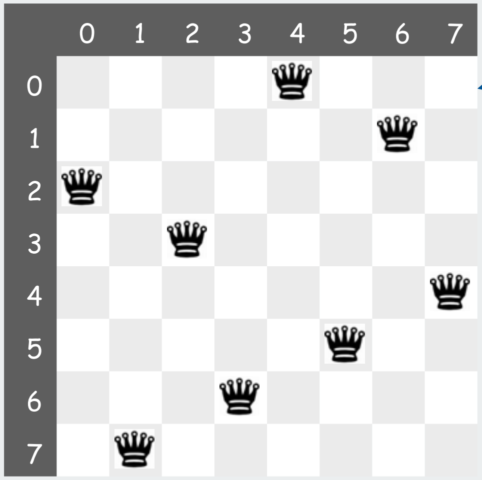



```
% java Rooks 25
```
millions of centuries

- permutations
- **backtracking**
- counting
- subsets
- paths in a graph

N-Queens problem

How many ways are there to place
N queens on an N-by-N board so that no queen can attack any other?

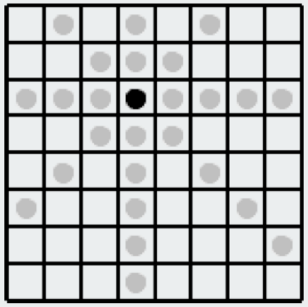


original problem: N = 8

```
int[] a = { 4, 6, 0, 2, 7, 5, 3, 1 };
```

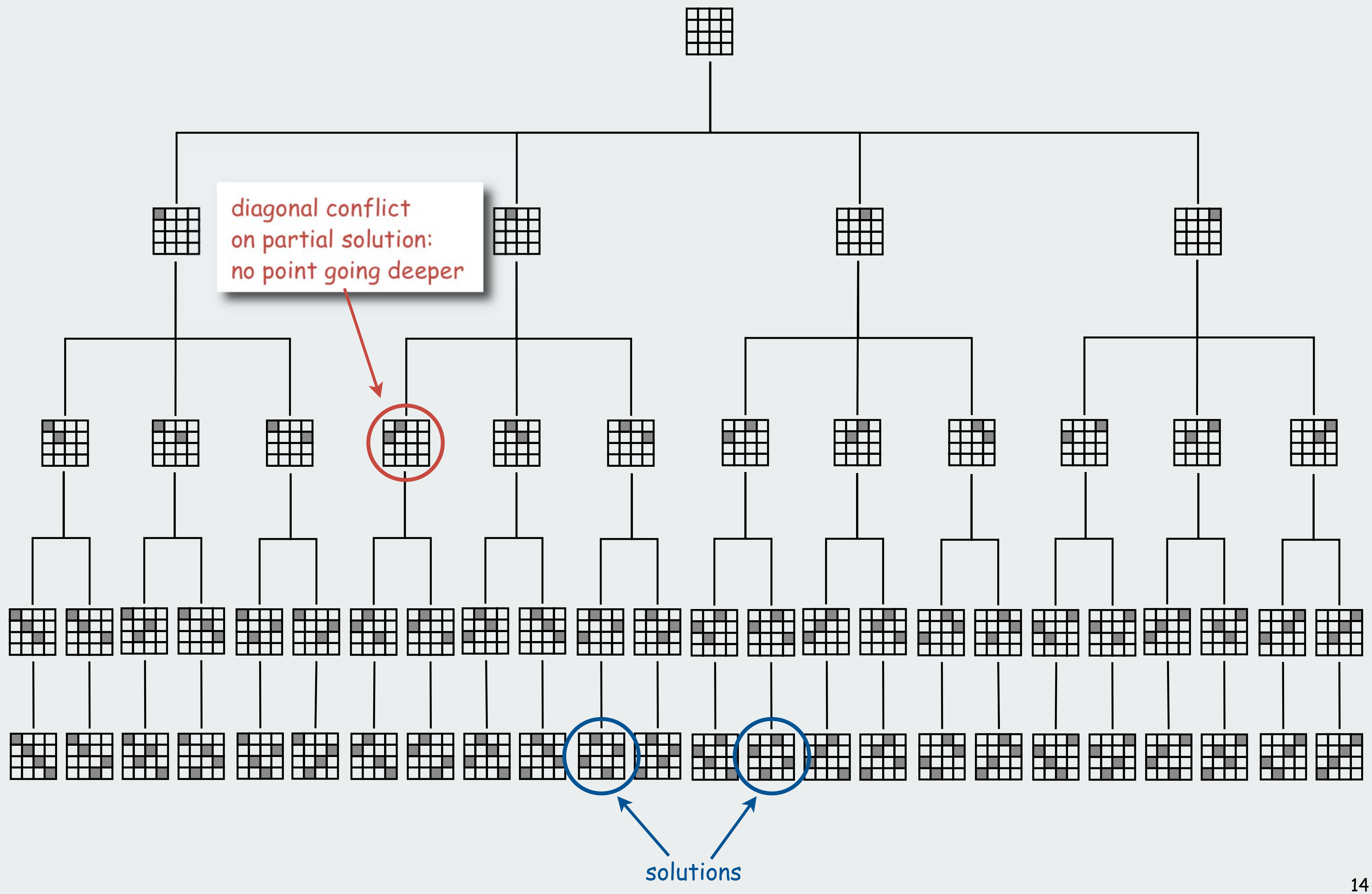
Representation. Same as for rooks:
represent solution as a permutation: `a[i]` = column of queen in row `i`.

Additional constraint: no diagonal attack is possible



Challenge: Enumerate (or even count) the solutions

4-Queens search tree



N Queens: Backtracking solution

Iterate through elements of search space.

- when there are N possible choices, make one choice and recur.
- if the choice is a dead end, backtrack to previous choice, and make next available choice.

Identifying dead ends allows us to prune the search tree

For N queens:

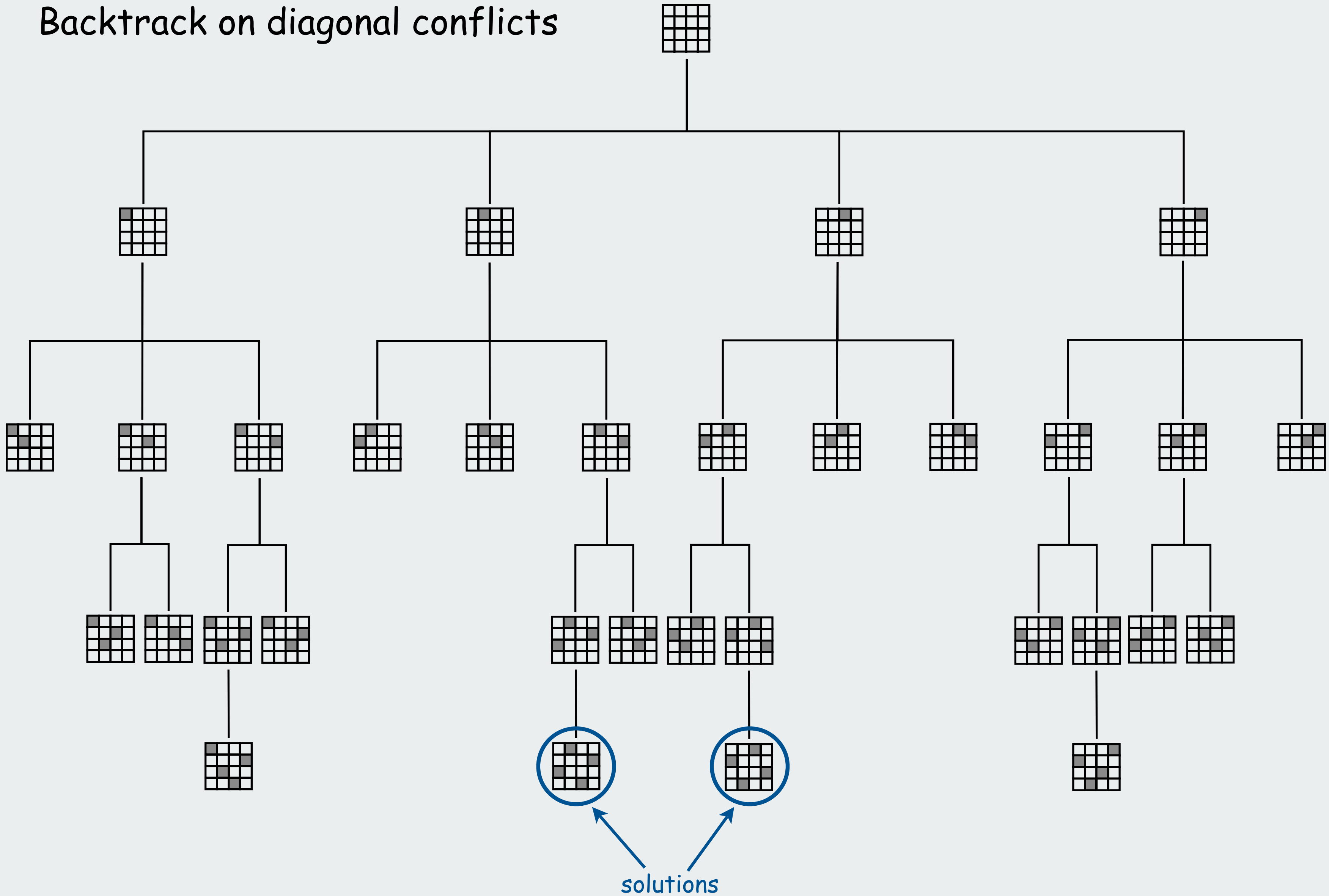
- dead end: a diagonal conflict
- pruning: backtrack and try next row when diagonal conflict found

In general, improvements are possible:

- try to make an "intelligent" choice
- try to reduce cost of choosing/backtracking

4-Queens Search Tree (pruned)

Backtrack on diagonal conflicts



N-Queens: Backtracking solution

```
private boolean backtrack(int k)
{
   for (int i = 0; i < k; i++)
   {
      if ((a[i] - a[k]) == (k - i)) return true;
      if ((a[k] - a[i]) == (k - i)) return true;
   }
   return false;
}

private void enumerate(int k)
{
   if (k == N)
   {
      process();
      return;
   }
   for (int i = k; i < N; i++)
   {
      exch(a, k, i);
      if (! backtrack(k)) enumerate(k+1);
      exch(a, k, i);
   }
}
```

stop enumerating if adding the n^{th} queen leads to a diagonal violation

```
% java Queens 4
1 3 0 2
2 0 3 1

% java Queens 5
0 2 4 1 3
0 3 1 4 2
1 3 0 2 4
1 4 2 0 3
2 0 3 1 4
2 4 1 3 0
3 1 4 2 0
3 0 2 4 1
4 1 3 0 2
4 2 0 3 1

% java Queens 6
1 3 5 0 2 4
2 5 1 4 0 3
3 0 4 1 5 2
4 2 0 5 3 1
```

N-Queens: Effectiveness of backtracking

Pruning the search tree leads to enormous time savings

N	2	3	4	5	6	7	8	9	10	11	12
Q(N)	0	0	2	10	4	40	92	352	724	2,680	14,200
N!	2	6	24	120	720	5,040	40,320	362,880	3,628,800	39,916,800	479,001,600

N	13	14	15	16
Q(N)	73,712	365,596	2,279,184	14,772,512
N!	6,227,020,800	87,178,291,200	1 ,307,674,368,000	20, 922,789,888,000

savings: factor of more than 1-million

N-Queens: How many solutions?

Answer to original question easy to obtain:

- add an instance variable to count solutions (initialized to 0)
- change **`process()`** to increment the counter
- add a method to return its value

```
% java Queens 4
2 solutions

% java Queens 8
92 solutions

% java Queens 16
14772512 solutions
```

Source: On-line encyclopedia of integer sequences, N. J. Sloane [sequence A000170]

N	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Q(N)	0	0	2	10	4	40	92	352	724	2,680	14,200	73,712	365,596	2,279,184

N	16	17	18	19		25
Q(N)	14,772,512	95,815,104	666,090,624	4,968,057,848	. . .	2, 207,893,435,808,350

took 53 years of CPU time (2005)

N-queens problem: back-of-envelope running time estimate

Hypothesis ??

```
% java Queens 13
73712 solutions
```
about a second

```
% java Queens 14
365596 solutions
```
about 7 seconds

```
% java Queens 15
2279184 solutions
```
about 49 seconds

```
% java Queens 16
14772512 solutions
```
about 360 seconds

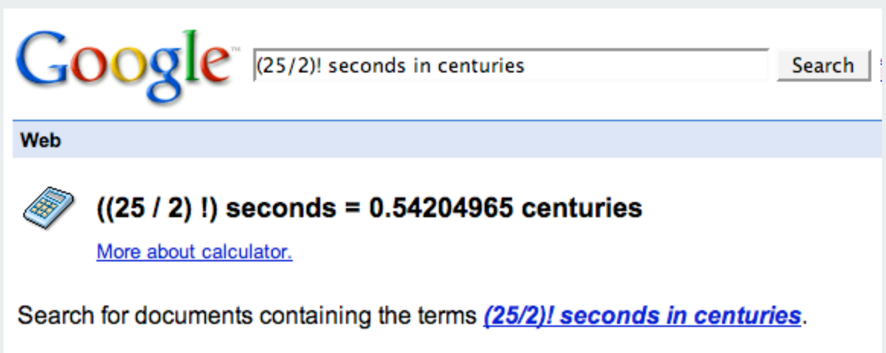
ratio

6.32

6.73

7.38

Hypothesis: Running time is about (N/2) ! seconds.




```
% java Queens 25
```
about 54 years ✓

- permutations
- backtracking
- **counting**
- subsets
- paths in a graph

Counting: Java Implementation

Problem: enumerate all N-digit base-R numbers
Solution: generalize binary counter in lecture warmup

enumerate N-digit base-R numbers

```
private static void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }

   for (int n = 0; n < R; n++)
   {
      a[k] = n;
      enumerate(k + 1);
   }
   a[k] = 0;
}
```

clean up not needed: Why?

enumerate binary numbers (from warmup)

```
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

clean up

0 0 0	1 0 0	2 0 0
0 0 1	1 0 1	2 0 1
0 0 2	1 0 2	2 0 2
0 1 0	1 1 0	2 1 0
0 1 1	1 1 1	2 1 1
0 1 2	1 1 2	2 1 2
0 2 0	1 2 0	2 2 0
0 2 1	1 2 1	2 2 1
0 2 2	1 2 2	2 2 2
0 2 0		
0 0 0		

example showing cleanups that zero out digits

Counting application: Sudoku

Problem:

Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

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

Remark: Natural generalization is NP-hard.

Counting application: Sudoku

Problem:

Fill 9-by-9 grid so that every row, column, and box contains each of the digits 1 through 9.

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

Solution: Enumerate all 81-digit base-9 numbers (with backtracking).

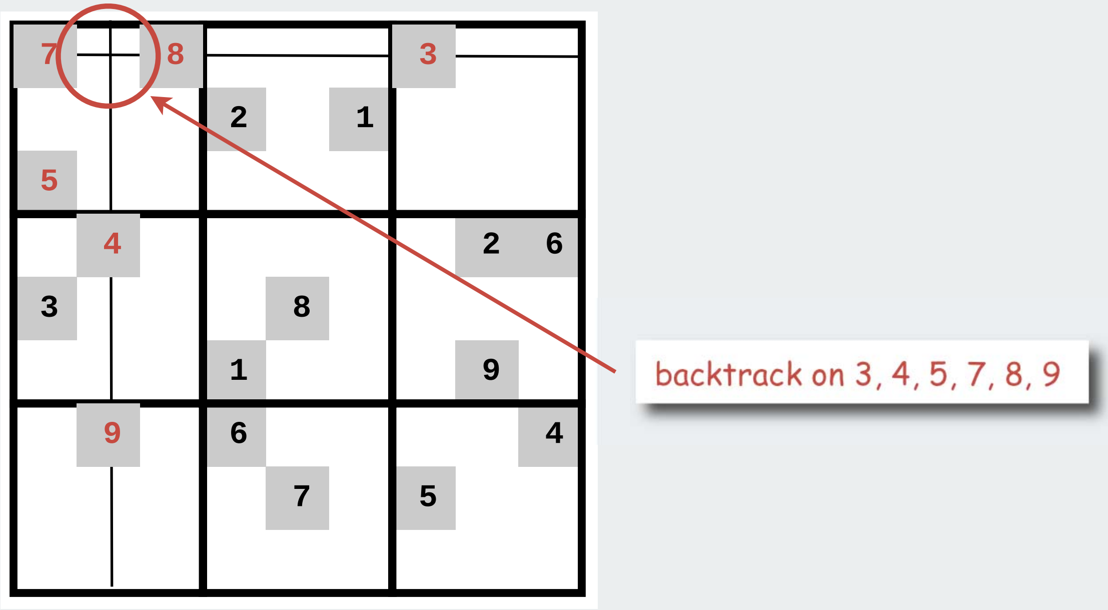
using digits 1 to 9 →

7		8					3		...
0	1	2	3	4	5	6	7	8	80

Sudoku: Backtracking solution

Iterate through elements of search space.

- For each empty cell, there are 9 possible choices.
- Make one choice and recur.
- If you find a conflict in row, column, or box, then backtrack.



Improvements are possible.

- try to make an "intelligent" choice
- try to reduce cost of choosing/backtracking

Sudoku: Java implementation

```
private static void solve(int cell)
{
   if (cell == 81)
   {  show(board); return;  }

   if (board[cell] != 0)
   {  solve(cell + 1); return;  }

   for (int n = 1; n <= 9; n++)
   {
      if (! backtrack(cell, n))
      {
         board[cell] = n;
         solve(cell + 1);
      }
   }

   board[cell] = 0;
}
```

already filled in

try all 9 possibilities

unless a Sudoku constraint is violated (see booksite for code)

clean up

```
...
int[81] board;
for (int i = 0; i < 81; i++)
   board[i] = StdOut.readInt();
Solver s = new Solver(board);
s.solve();
...
```

```
% more board.txt
7 0 8 0 0 0 3 0 0
0 0 0 2 0 1 0 0 0
5 0 0 0 0 0 0 0 0
0 4 0 0 0 0 0 2 6
3 0 0 0 8 0 0 0 0
0 0 0 1 0 0 0 9 0
0 9 0 6 0 0 0 0 4
0 0 0 0 7 0 5 0 0
0 0 0 0 0 0 0 0 0

% java Solver
7 2 8 9 4 6 3 1 5
9 3 4 2 5 1 6 7 8
5 1 6 7 3 8 2 4 9
1 4 7 5 9 3 8 2 6
3 6 9 4 8 2 1 5 7
8 5 2 1 6 7 4 9 3
2 9 3 6 1 5 7 8 4
4 8 1 3 7 9 5 6 2
6 7 5 8 2 4 9 3 1
```

Works remarkably well (plenty of constraints). Try it!

- permutations
- backtracking
- counting
- **subsets**
- paths in a graph

Enumerating subsets: natural binary encoding

Given n items, enumerate all 2^n subsets.
- count in binary from 0 to 2^n - 1.
- bit i represents item i
- if 0, in subset; if 1, not in subset

i	binary	subset	complement
0	0 0 0 0	empty	4 3 2 1
1	0 0 0 1	1	4 3 2
2	0 0 1 0	2	4 3 1
3	0 0 1 1	2 1	4 3
4	0 1 0 0	3	4 2 1
5	0 1 0 1	3 1	4 2
6	0 1 1 0	3 2	4 1
7	0 1 1 1	3 2 1	4
8	1 0 0 0	4	3 2 1
9	1 0 0 1	4 1	3 2
10	1 0 1 0	4 2	3 1
11	1 0 1 1	4 2 1	3
12	1 1 0 0	4 3	2 1
13	1 1 0 1	4 3 1	2
14	1 1 1 0	4 3 2	1
15	1 1 1 1	4 3 2 1	empty

Enumerating subsets: natural binary encoding

Given N items, enumerate all 2^N subsets.

- count in binary from 0 to $2^N - 1$.
- maintain `a[i]` where `a[i]` represents item `i`
- if `0`, `a[i]` in subset; if `1`, `a[i]` not in subset

Binary counter from warmup does the job

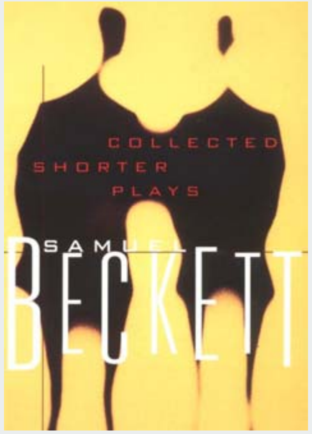
```
private void enumerate(int k)
{
  if (k == N)
  {  process(); return;  }
  enumerate(k+1);
  a[k] = 1;
  enumerate(k+1);
  a[k] = 0;
}
```

Digression: Samuel Beckett play

Quad. Starting with empty stage, 4 characters enter and exit one at a time, such that each subset of actors appears exactly once.

code	*subset*	*move*
0 0 0 0	*empty*	
0 0 0 1	1	enter 1
0 0 1 1	2 1	enter 2
0 0 1 0	2	exit 1
0 1 1 0	3 2	enter 3
0 1 1 1	3 2 1	enter 1
0 1 0 1	3 1	exit 2
0 1 0 0	3	exit 1
1 1 0 0	4 3	enter 4
1 1 0 1	4 3 1	enter 1
1 1 1 1	4 3 2 1	enter 2
1 1 1 0	4 3 2	exit 1
1 0 1 0	4 2	exit 3
1 0 1 1	4 2 1	enter 1
1 0 0 1	4 1	exit 2
1 0 0 0	4	exit 1

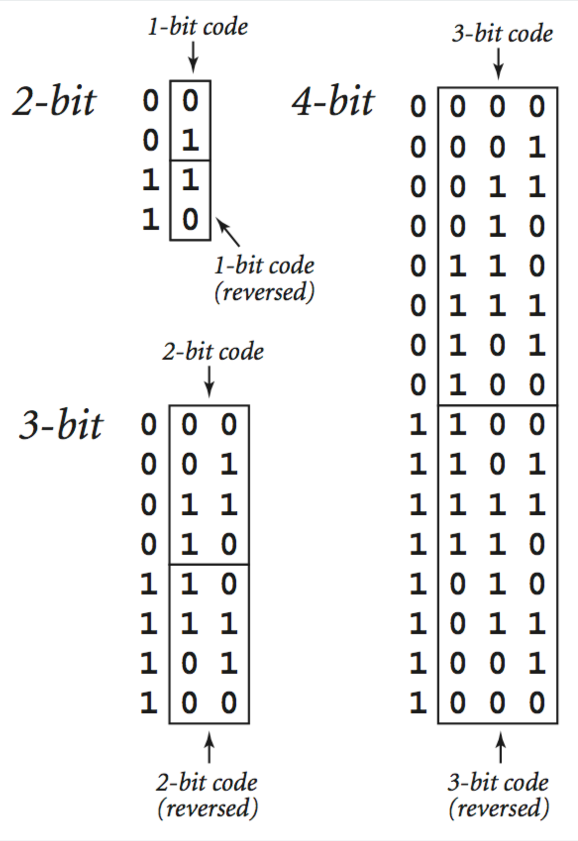
ruler function



Binary reflected gray code

The n-bit binary reflected Gray code is:

- the (n-1) bit code with a 0 prepended to each word, followed by
- the (n-1) bit code in reverse order, with a 1 prepended to each word.



Beckett: Java implementation

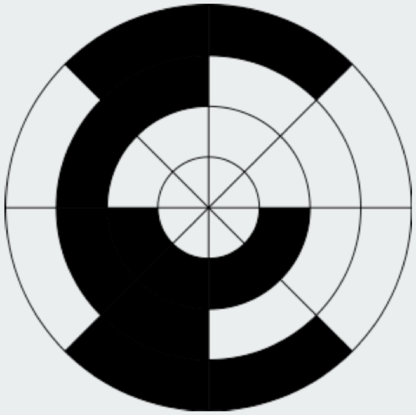
```
public static void moves(int n, boolean enter)
{
   if (n == 0) return;
   moves(n-1, true);
   if (enter) System.out.println("enter " + n);
   else       System.out.println("exit  " + n);
   moves(n-1, false);
}
```

```
% java Beckett 4
enter 1
enter 2
exit  1
enter 3
enter 1
exit  2
exit  1
enter 4
enter 1
enter 2
exit  1
exit  3
enter 1
exit  2
exit  1
```

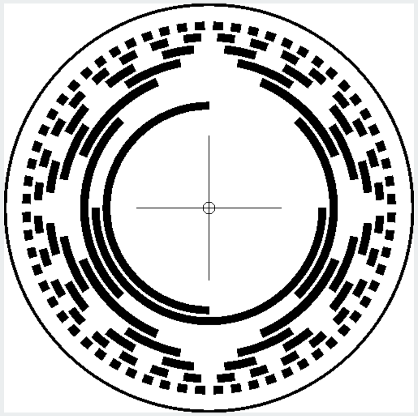
stage directions
for 3-actor play
`moves(3, true)`

reverse stage directions
for 3-actor play
`moves(3, false)`

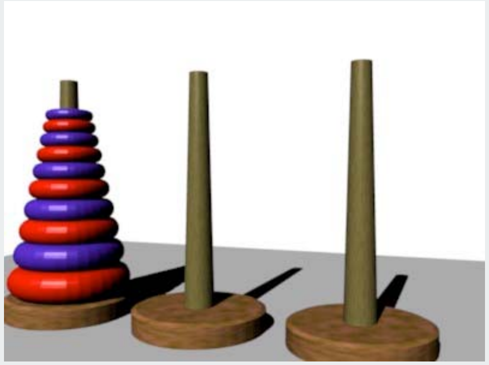
More Applications of Gray Codes



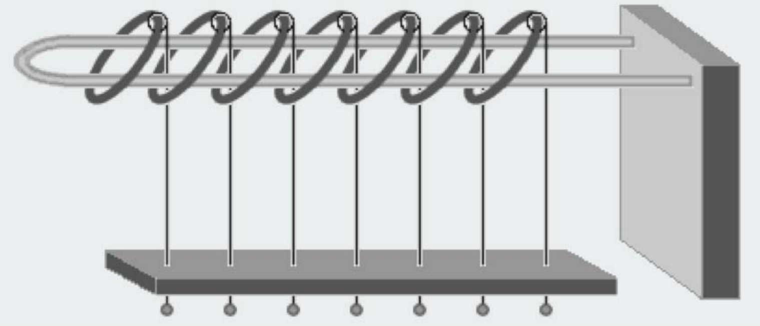
3-bit rotary encoder



8-bit rotary encoder



Towers of Hanoi



Chinese ring puzzle

Enumerating subsets using Gray code

Two simple changes to binary counter from warmup:

- flip `a[k]` instead of setting it to `1`
- eliminate cleanup

Gray code enumeration

```
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }
   enumerate(k+1);
   a[k] = 1 - a[k];
   enumerate(k+1);
}
```

standard binary (from warmup)

```
private void enumerate(int k)
{
   if (k == N)
   {  process(); return;  }
   enumerate(k+1);
   a[k] = 1;
   enumerate(k+1);
   a[k] = 0;
}
```

clean up

000
001
011
010
110
111
101
100

000
001
010
011
100
101
110
111

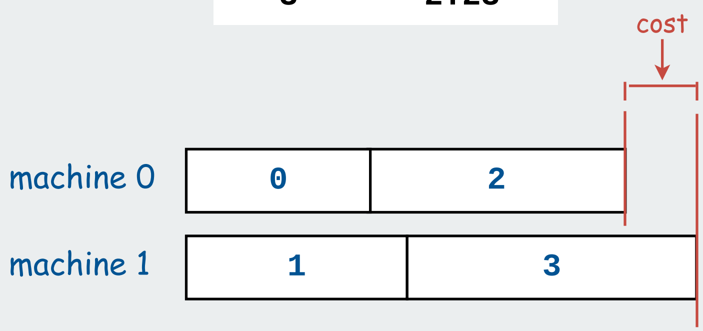
Advantage (same as Beckett): only one item changes subsets

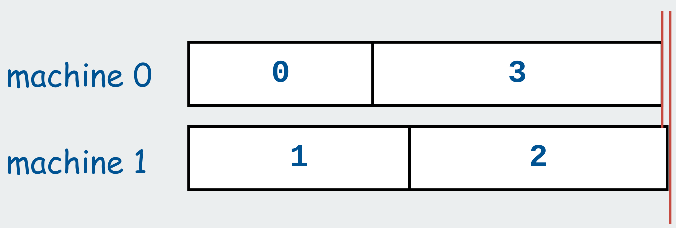
Scheduling

Scheduling (set partitioning). Given n jobs of varying length, divide among two machines to minimize the time the last job finishes.

or, equivalently, difference between finish times

job	length
0	1.41
1	1.73
2	2.00
3	2.23






```
public double[] finish(int[] a)
{
    double[] time = new double[2];
    time[0] = 0.0; time[1] = 0.0;
    for (int i = 0; i < N; i++)
        time[a[i]] += jobs[i];
    return time;
}

private double cost(int[] a)
{
    double[] time = finish(a);
    return Math.abs(time[0] - time[1]);
}
```

i	a[]	time[0]	time[1]
	0 1 1 0	1.41	0
0	0 1 1 0	1.41	0
1	0 1 1 0	1.41	1.73
2	0 1 1 0	1.41	3.73
3	0 1 1 0	3.64	3.73
		3.64	3.73

cost: .09

Remark: NP-hard.

Scheduling (full implementation)

```
public class Scheduler
{
   int N;           // Number of jobs.
   int[] a;         // Subset assignments.
   int[] b;         // Best assignment.
   double[] jobs;   // Job lengths.

   public Scheduler(double[] jobs)
   {
      this.N = jobs.length;;
      this.jobs = jobs;
      a = new int[N];
      b = new int[N];
      for (int i = 0; i < N; i++)
          a[i] = 0;
      for (int i = 0; i < N; i++)
          b[i] = a[i];
      enumerate(0);
   }

   public int[] best()
   {  return b;  }


   private void enumerate(int k)
   { /* Gray code enumeration. */   }

   private void process()
   {
     if (cost(a) < cost(b))
       for (int i = 0; i < N; i++)
         b[i] = a[i];
   }

   public static void main(String[] args)
   { /* Create Scheduler, print result. */  }
}
```

trace of
`% java Scheduler 4 < jobs.txt`

a[] finish times cost

```
0   0   0   0      7.38    0.00
0   0   0   1      5.15    2.24     2.91
0   0   1   1      3.15    4.24     1.09
0   0   1   0      5.38    2.00
0   1   1   0      3.65    3.73     0.08
0   1   1   1      1.41    5.97
0   1   0   1      3.41    3.97
0   1   0   0      5.65    1.73
1   1   0   0      4.24    3.15
1   1   0   1      2.00    5.38
1   1   1   1      0.00    7.38
1   1   1   0      2.24    5.15
1   0   1   0      3.97    3.41
1   0   1   1      1.73    5.65
1   0   0   1      3.73    3.65
1   0   0   0      5.97    1.41
   MACHINE 0      MACHINE 1
  1.4142135624
                1.7320508076
                2.0000000000
  2.2360679775
-----------------------------
  3.6502815399  3.7320508076
```

Scheduling (larger example)

```
java SchedulerEZ 24 < jobs.txt
 MACHINE 0       MACHINE 1
1.4142135624
1.7320508076
                2.0000000000
2.2360679775
2.4494897428
                2.6457513111
                2.8284271247
                3.0000000000
3.1622776602
                3.3166247904
                3.4641016151
                3.6055512755
                3.7416573868
3.8729833462
                4.0000000000
4.1231056256
                4.2426406871
4.3588989435
                4.4721359550
4.5825756950
4.6904157598
4.7958315233
4.8989794856
                5.0000000000
------------------------------
 42.3168901295 42.3168901457
```

cost < 10^{-8}

Large number of subsets leads to remarkably low cost

Scheduling: improvements

Many opportunities (details omitted)

- fix last job on machine 0 (quick factor-of-two improvement)
- backtrack when partial schedule cannot beat best known (check total against goal: half of total job times)

```
private void enumerate(int k)
{
  if (k == N-1)
  {  process(); return;  }
  if (backtrack(k)) return;
  enumerate(k+1);
  a[k] = 1 - a[k];
  enumerate(k+1);
}
```

- process all 2^k subsets of last k jobs, keep results in memory, (reduces time to 2^{N-k} when 2^k memory available).

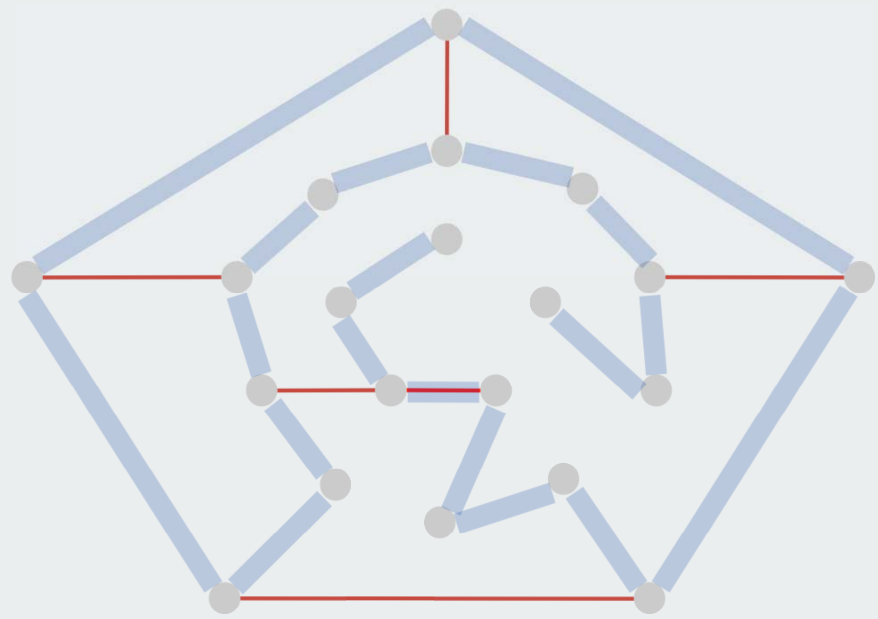
Backtracking summary

N-Queens : permutations with backtracking
Soduko : counting with backtracking
Scheduling: subsets with backtracking

- permutations
- backtracking
- counting
- subsets
- **paths in a graph**

Hamilton Path

Hamilton path. Find a simple path that visits every vertex exactly once.

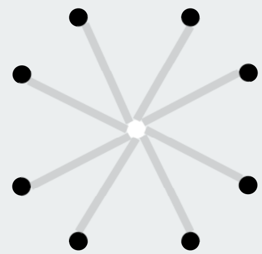


Remark. Euler path easy, but Hamilton path is NP-complete.

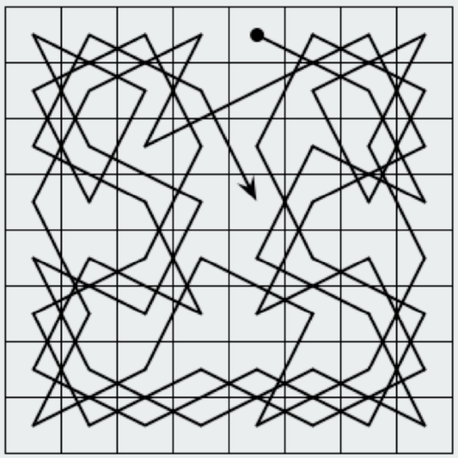
visit every edge exactly once

Knight's Tour

Knight's tour. Find a sequence of moves for a knight so that, starting from any square, it visits every square on a chessboard exactly once.



legal knight moves



a knight's tour

Solution. Find a Hamilton path in knight's graph.

Hamilton Path: Backtracking Solution

Backtracking solution. To find Hamilton path starting at v:

- Add v to current path.
- For each vertex w adjacent to v
  find a simple path starting at w using all remaining vertices
- Remove v from current path.

How to implement?
Add cleanup to DFS (!!)

Hamilton Path: Java implementation

```
public class HamiltonPath
{
   private boolean[] marked;
   private int count;

   public HamiltonPath(Graph G)
   {
      marked = new boolean[G.V()];
      for (int v = 0; v < G.V(); v++)
         dfs(G, v, 1);
      count = 0;
   }

   private void dfs(Graph G, int v, int depth)
   {
      marked[v] = true;

      if (depth == G.V()) count++;

      for (int w : G.adj(v))
         if (!marked[w]) dfs(G, w, depth+1);

      marked[v] = false;
  }
}
```

also need code to count solutions (path length = V)

clean up

Easy exercise: Modify this code to find and print the longest path

The Longest Path

Recorded by Dan Barrett in 1988 while a student at Johns Hopkins during a difficult algorithms final.

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!

If you said P is NP tonight,
There would still be papers left to write,
I have a weakness,
I'm addicted to completeness,
And I keep searching for the longest path.

The algorithm I would like to see
Is of polynomial degree,
But it's elusive:
Nobody has found conclusive
Evidence that we can find a longest path.

I have been hard working for so long.
I swear it's right, and he marks it wrong.
Some how I'll feel sorry when it's done: GPA 2.1
Is more than I hope for.

Garey, Johnson, Karp and other men (and women)
Tried to make it order N log N.
Am I a mad fool
If I spend my life in grad school,
Forever following the longest path?

Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path!
Woh-oh-oh-oh, find the longest path.